Forward and Backward Private Searchable Encryption from Constrained Cryptographic Primitives

Raphael Bost[*] Brice Minaud[†] Olga Ohrimenko[‡]

Abstract

Using dynamic Searchable Symmetric Encryption, a user with limited storage resources can securely outsource a database to an untrusted server, in such a way that the database can still be searched and updated efficiently. For these schemes, it would be desirable that updates do not reveal any information *a priori* about the modifications they carry out, and that deleted results remain inaccessible to the server *a posteriori*. If the first property, called *forward privacy*, has been the main motivation of recent works, the second one, *backward privacy*, has been overlooked.

In this paper, we study for the first time the notion of backward privacy for searchable encryption. After giving formal definitions for different flavors of backward privacy, we present several schemes achieving both forward and backward privacy, with various efficiency trade-offs.

Our constructions crucially rely on primitives such as constrained pseudo-random functions and puncturable encryption schemes. Using these advanced cryptographic primitives allows for a fine-grained control of the power of the adversary, preventing her from evaluating functions on selected inputs, or decrypting specific ciphertexts. In turn, this high degree of control allows our SSE constructions to achieve the stronger forms of privacy outlined above. As an example, we present a framework to construct forward-private schemes from range-constrained pseudo-random functions.

Finally, we provide experimental results for implementations of our schemes, and study their practical efficiency.

1 Introduction

Symmetric Searchable Encryption (SSE) enables a client to outsource the storage of private data to an untrusted server, while retaining the ability to issue search queries over the outsourced data. *Dynamic* SSE schemes add the ability for the client to update the outsourced database, inserting and possibility deleting entries remotely. All the while, the design of the scheme should ensure that the server is able to infer as little as possible about the content of the database, or even the content of the queries it processes.

At the core of SSE schemes are trade-offs between efficiency, such as storage requirements, bandwidth or latency, and the degree to which the scheme protects the content of the client's data against a curious (or malicious) server. The latter is captured by the notion of *leakage functions* that restrict the type of information leaked to the server while processing search or update queries.

Since the inception of searchable encryption, tremendous progress has been made toward efficient solutions yielding high throughput, low latency, and more expressive queries [CJJ$^+$13, CJJ$^+$14, MM17]. Amid a growing awareness of privacy concerns however, a different line of work has uncovered devastating and fairly generic attacks against many SSE schemes [CGPR15, ZKP16]. Such leakage-abuse attacks do not contradict the security claims of the targeted SSE schemes, but show how seemingly benign leakage functions can be exploited to reveal a considerable amount of information in practice.

[*]Direction Générale de l'Armement - Maîtrise de l'Information & Université de Rennes 1, France. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the DGA or the French Government. email: raphael_bost@alumni.brown.edu

[†]Royal Holloway, University of London, UK. email: brice.minaud@gmail.com

[‡]Microsoft Research, Cambridge, UK. email: oohrim@microsoft.com

Table 1 – Comparison with prior work. N is the number of keyword/document pairs in the database, K the number of distinct keywords, and D the number of documents. n_w is the size of the search result set for keyword w, a_w is the number of entries matching w inserted in total, while d_w is the number of deleted entries matching w (and $n_w = a_w - d_w$). The RT column stands for the number of roundtrips in the search protocol. FP (resp. BP) stands for forward (resp. backward) privacy. We denote different levels of backward privacy with I, II, and III, where I is the strongest level (see Section 4.2 for details). The notation $\widetilde{O}$ hides polylog factors.

Scheme	Computation		Communication			Client Storage	FP	BP
	Search	Update	Search	RT	Update			
Π^{dyn} [CJJ+14]	$O(a_w)$	$O(1)$	$O(n_w)$	1	$O(1)$	$O(1)$	✗	-
SPS [SPS14]	$O\left(\min\left\{\begin{matrix} a_w + \log N \\ n_w \log^3 N \end{matrix}\right\}\right)$	$O(\log^2 N)$	$O(n_w + \log N)$	1	$O(\log N)$	$O(N^\alpha)$	✓	-
TWORAM [GMP16]	$\widetilde{O}\left(n_w \log N + \log^3 N\right)$	$\widetilde{O}(\log^2 N)$	$\widetilde{O}(n_w \log N + \log^3 N)$	2	$\widetilde{O}(\log^3 N)$	$O(1)$	✓	-
$\Sigma o\phi o\varsigma$ [Bos16]	$O(a_w)$	$O(1)$	$O(n_w)$	1	$O(1)$	$O(K \log D)$	✓	-
ARX [PBP16]	$O(a_w)$	$O(\log a_w)$	$O(n_w + \log a_w)$	1	$O(1)$	$O(K \log D)$	✓	-
Moneta § 4.3	$\widetilde{O}\left(a_w \log N + \log^3 N\right)$	$\widetilde{O}(\log^2 N)$	$\widetilde{O}(a_w \log N + \log^3 N)$	3	$\widetilde{O}(\log^3 N)$	$O(1)$	✓	I
Fides § 4.4	$O(a_w)$	$O(1)$	$O(a_w)$	2	$O(1)$	$O(K \log D)$	✓	II
Diana § 5.2	$O(a_w)$	$O(\log a_w)$	$O(n_w + \log a_w)$	1	$O(1)$	$O(K \log D)$	✓	-
Diana$_{\mathsf{del}}$ § 5.3	$O(a_w)$	$O(\log a_w)$	$O(n_w + d_w \log a_w)$	2	$O(1)$	$O(K \log D)$	✓	III
Janus § 6	$O(n_w \cdot d_w)$	$O(1)$	$O(n_w)$	1	$O(1)$	$O(K \log D)$	✓	III

Forward privacy (also known as forward security) is an important property of searchable encryption schemes that mitigates these attacks by ensuring that newly updated entries cannot be related to previous search results. Notably, forward-private schemes prevent the most powerful versions of the recent injection attacks by Zhang *et al.* [ZKP16]. Another natural notion of privacy is that of *backward privacy*: search queries should not leak matching entries after they have been deleted. However, besides being mentioned by Stefanov *et al.* [SPS14], it is almost not discussed in the literature.

Our contribution. In this work, we realize single-keyword SSE constructions from constrained and puncturable primitives. By leveraging the fine-grained control afforded by this type of primitives, we are able to build (1) a very efficient forward-secure scheme; and (2) a scheme that achieves both forward privacy and a weak form of backward privacy. For both schemes, we define and prove a general framework to build forward-secure SSE from the abstract SSE primitive; and then propose and study a concrete scheme by instantiating the framework with a specific choice of the underlying primitive. In the process, we also investigate the notion of backward privacy, providing formal definitions and a generic construction. Finally, we provide experimental results for implementations of our schemes. In more detail, our contributions are as follows.

(1) We propose formal definitions for several forms of backward privacy, which up to now had only been treated informally. We also describe a simple and generic way to achieve backward privacy from any forward private SSE scheme at the cost of an extra roundtrip per search query, and two instantiations, Moneta and Fides.

(2) We define the FS-RCPRF framework, which builds a single-keyword forward-private SSE scheme from any constrained pseudo-random function (CPRF) compatible with range constraints. By instantiating the CPRF with the classic construction by Goldreich, Goldwasser and Micali [GGM84], we obtain Diana, a forward-secure SSE scheme with very low computational and bandwidth overhead—on some data sets, Diana performs up to 10 times faster than recent schemes from the literature achieving the same leakage. Note that Diana is very similar to the ARX-EQ construction [PBP16]. We also show how we can modify Diana into a two-roundtrips backward-private scheme Diana$_{\mathsf{del}}$.

(3) Finally, we describe Janus, a framework for constructing a forward-secure SSE scheme that also achieves a weak form of backward privacy; namely, search queries do not leak entries that match the query

after the entry has been deleted. The Janus framework requires a puncturable encryption scheme with a particular incremental update property, which can be instantiated by the Green-Miers puncturable encryption scheme [GM15].

To the best of our knowledge, Fides, $\mathsf{Diana}_{\mathsf{del}}$ and Janus are the first schemes not based on oblivious RAM to achieve backward (and forward) privacy. Moreover, Janus is the only existing single-roundtrip forward and backward-private scheme.

A comparison of our schemes with prior work is provided in Table 1. Beside the schemes themselves, we believe this work draws a new connection between constrained primitives and searchable encryption, which from the perspective of SSE schemes means new construction techniques, and from the perspective of constrained or puncturable primitives, new applications.

2 Related Work

Searchable Encryption. Song *et al.* [SWP00] first introduced SSE. The modern security definitions were developed by Curtmola *et al.* [CGKO06]. They introduced the idea of leakage, and designed the first reversed-index-based SSE construction, achieving optimal search complexity. Note that SSE is a particular case of structured encryption, as defined by Chase and Kamara [CK10], focused on multi-maps (*a.k.a.* T-Sets or reversed index).

Even though the dynamic setting had been studied earlier, Kamara and Papamanthou [KP13] designed the first sublinear dynamic scheme. Cash *et al.* [CJJ$^+$14] constructed a dynamic scheme optimized for large datasets.

Forward privacy was introduced by Stefanov *et al.* in [SPS14]. In that paper, the authors present an ORAM-inspired forward-private SSE construction. Their construction also deals with deletion in an elegant way, as it allows the server to skip deleted entries. However, this was only designed to improve the performance of the scheme, rather than its security. In [Bos16], Bost formally defined forward privacy and designed an insertion-only SSE scheme with optimal search and update complexity, based on asymmetric cryptography (trapdoor permutations). The motivation for studying forward security came from file injection attacks on SSE [ZKP16].

In order to achieve the highest security guarantees, SSE can be constructed using Oblivious RAM components [GO96, GMP16]. Unfortunately the overhead of ORAM is too high for a practical SSE scheme [Nav15].

Several results propose SSE schemes with expressive search queries: Cash *et al.* [CJJ$^+$13] considered conjunctive queries; Kamara and Moataz [KM17] built a scheme for disjunctive queries; while graph encryption was studied by Chase and Kamara [CK10] and Meng *et al.* [MKNK15].

Constrained cryptographic primitives. Constrained pseudorandom functions were concurrently introduced in [BW13, KPTZ13, BGI14], and applied to broadcast encryption, identity-based key-exchange, or SNARGs. One application considered by Kiayias *et al.* [KPTZ13] was actually searchable encryption, but only for performance reasons: the constrained PRF is used to batch queries. Instead of transmitting to the server many pseudo-randomly generated trapdoors, the client would transmit a constrained key allowing for the generation of the trapdoors by the server.

Since then, new constrained PRFs have been developed, with the ability to support richer constraint spaces [HKW15, CC17]. Unfortunately, many of these new constructions rely on indistinguishability obfuscation or similar techniques, and hence are not yet practical. In this work, we only require the existence of cryptographic pairings for puncturable encryption, and pseudo-random functions.

Building on non-monotonic attribute-based encryption [OSW07], Green and Miers [GM15] proposed puncturable encryption as a way to achieve forward secrecy for instant messaging. Their scheme modifies the secret key every time a message is received, so that from then on the modified key can no longer decrypt that message. Thus, in the event of a key compromise, old messages remain safe.

Secure deletion. In this paper, we will use puncturable encryption to securely delete entries in an encrypted database. Indeed, Green and Miers [GM15] mention secure deletion as another application of their work.

Boneh and Lipton [BL96] were the first to suggest using cryptography to erase information. These cryptographic solutions were implemented for filesystems on flash drives [RCB12]. Secure deletion and history independence properties were also considered in oblivious RAM literature [RAC16].

3 Background

In the paper, λ is the security parameter and $\mathrm{negl}(\lambda)$ denotes a negligible function in the security parameter.

Unless specified explicitly, the symmetric keys are strings of λ bits, and the key generation algorithm uniformly samples a key in $\{0,1\}^\lambda$. We only consider (probabilistic) algorithms and protocols running in time polynomial in the security parameter λ. In particular, adversaries are probabilistic polynomial-time (PPT) algorithms.

For a finite set X, $x \overset{\$}{\leftarrow} X$ means that x is sampled uniformly from X.

3.1 Constrained Pseudorandom Functions

The idea of *constrained PRFs* (CPRFs) has been introduced in concurrent work by Boneh and Waters, Boyle *et al.*, and Kiayias *et al.* [BW13, BGI14, KPTZ13]. A constrained PRF is associated with a family of boolean circuits $\mathcal{C}$. The holder of the master PRF key is able to compute a *constrained key* K_C corresponding to a circuit $C \in \mathcal{C}$; the constrained key K_C allows evaluation of the PRF only on inputs x for which $C(x) = 1$.

More formally, a *constrained PRF* F with respect to a circuit family $\mathcal{C}$ is a mapping $F : \{0,1\}^\lambda \times X \to Y$ (the PRF proper), together with a pair of algorithms ($F.\mathsf{Constrain}$, $F.\mathsf{Eval}$), defined as follows.

- $F.\mathsf{Constrain}(K, C)$ is a PPT algorithm taking as input a key $K \in \{0,1\}^\lambda$ and a circuit $C \in \mathcal{C}$. It outputs a constrained key K_C.
- $F.\mathsf{Eval}(K_C, x)$ is a deterministic polynomial-time algorithm taking as input a constrained key K_C for circuit C, and $x \in X$. It outputs $y \in Y$.

Wherever this does not result in ambiguity, we may leave out Eval and write $F.\mathsf{Eval}(K_C, x)$ as $F(K_C, x)$.

Correctness. A CPRF F is correct iff $C(x) = 1$ implies $F(K, x) = F.\mathsf{Eval}(K_C, x)$, where $K_C = F.\mathsf{Constrain}(K, C)$, for all K, x, and $C \in \mathcal{C}$.

Security. The security game describing the security of a CPRF has three phases.

Setup Phase The challenger randomly picks a key $K \overset{\$}{\leftarrow} \{0,1\}^\lambda$ and a bit $b \overset{\$}{\leftarrow} \{0,1\}$.

Query Phase The adversary can adaptively query the oracles:

$Eval(x)$ The challenger returns $F(K, x)$;

$Constrain(C)$ The challenger returns $F.\mathsf{Constrain}(K, C)$;

$Challenge(x)$ If $b = 0$ the challenger outputs $F(K, x)$, otherwise he returns a uniform element in Y.

Guess Phase The adversary outputs a guess b' of b.

Let E be the set of evaluation queries, Z the set of challenge queries, L the set of constrained key queries. The adversary wins the game if $b = b'$ and $E \cap Z = \emptyset$ and $C(z) = 0\ \forall C \in L$ and $z \in Z$.

3.2 Bilinear Maps

Let $\mathbb{G}$ and $\mathbb{G}_T$ be two cyclic groups of prime order p, g be a generator of $\mathbb{G}$ and $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$ be such that

- e is bilinear: for all $x, y \in \mathbb{G}$, $a, b \in \mathbb{Z}_p$, $e(x^a, y^b) = e(x, y)^{ab}$;
- e is non-degenerate: $e(g, g) \neq 1$.

We consider $\mathbb{G}$, $\mathbb{G}_T$ and e such that the group operations in $\mathbb{G}$ and $\mathbb{G}_T$, and the bilinear map e are all efficiently computable. The scheme presented in this work using pairings needs the Decisional Bilinear Diffie-Hellman (DBDH) and Decisional Bilinear Diffie-Hellman Inversion (DBDHI) to hold (*cf.* [BB04]).

3.3 Symmetric Searchable Encryption

The *database* DB on which we wish to perform search queries is defined as: $\mathsf{DB} = \{(\mathsf{ind}_i, \mathsf{W}_i) : 1 \leq i \leq D\}$, with $\mathsf{ind}_i \in \{0,1\}^\ell, \mathsf{W}_i \subseteq \{0,1\}^*$, and where ind_i are distinct *document indices*, represented by ℓ-bit strings, and W_i is a set of *keywords* matching document ind_i, represented by binary strings of arbitrary length. Note that we identify documents with their indices. In addition, let us define:

$$\begin{aligned} \mathsf{W} &= \cup_{i=1}^{D} \mathsf{W}_i \text{ the set of keywords;} \\ K &= |\mathsf{W}| \text{ the number of keywords;} \\ D &= |\mathsf{DB}| \text{ the number of documents;} \\ N &= \textstyle\sum_{i=1}^{D} |\mathsf{W}_i| \text{ the number of document/keyword pairs.} \end{aligned}$$

Finally, let $\mathsf{DB}(w)$ denote the set of documents containing keyword w, *i.e.* $\mathsf{DB}(w) = \{\mathsf{ind}_i | w \in \mathsf{W}_i\}$.

A *dynamic searchable encryption scheme* Σ is a triple $(\mathsf{Setup}, \mathsf{Search}, \mathsf{Update})$ consisting of one algorithm and two protocols between a client and a server:

- $\mathsf{Setup}(\mathsf{DB})$ is a probabilistic algorithm that takes as input the initial database DB. It outputs a triple $(\mathsf{EDB}, K_\Sigma, \sigma)$, where K_Σ is the master secret key, EDB is an encrypted database, and σ is the client's state.
- $\mathsf{Search}(K_\Sigma, q, \sigma; \mathsf{EDB}) = (\mathsf{Search}_C(K_\Sigma, q, \sigma), \mathsf{Search}_S(\mathsf{EDB}))$ is a protocol between the client whose input is the master secret key K_Σ, the client's internal state σ, and a search query q; and the server whose input is the encrypted database EDB. In this paper, we only consider search queries restricted to a single keyword w.
- $\mathsf{Update}(K_\Sigma, \sigma, \mathsf{op}, \mathsf{in}; \mathsf{EDB}) = (\mathsf{Update}_C(K_\Sigma, \sigma, \mathsf{op}, \mathsf{in}), \mathsf{Update}_S(\mathsf{EDB}))$ is a protocol between the client whose input is K_Σ and σ as above, and an operation op with its input in, where in is parsed as an index ind and a set W of keywords; and the server with input EDB. The update operations are taken from the set $\{\mathsf{add}, \mathsf{del}\}$, meaning, respectively, the addition and the deletion of a document/keyword pair.

An SSE scheme is said to be *correct* if the search protocol returns the correct result for every query, except with negligible probability. We refer to [CJJ+14] for a formal definition of correctness.

Security The security of an SSE scheme expresses the fact that the server should learn as little as possible about the content of the database and queries. More precisely, we do not want the adversary to learn anything beyond some explicit leakage. This is typically captured using a real-world versus ideal-world formalization [CGKO06, KPR12, CJJ+14]. A *leakage function* is used to express the information leaked to the adversary by each SSE operation. Formally, the model is parametrized by the (stateful) leakage function $\mathcal{L} = (\mathcal{L}^{\mathsf{Stp}}, \mathcal{L}^{\mathsf{Srch}}, \mathcal{L}^{\mathsf{Updt}})$, whose components correspond respectively to the Setup, Search, and Update operations. The security model expresses the fact that whenever the client triggers one of these operations, the adversary learns no more than the output of the corresponding leakage function.

Formally, the adversary's task is to distinguish between a real word SSEReal and an ideal world SSEIdeal. The adversary fully controls the client, in the sense that she can trigger Setup, then Search and

Update queries at will, with parameters of her choosing. She then observes the execution of the scheme from the point of view of the server. That is, the adversary is able to observe the full transcript of each operation, *i.e.* the full content of the communication between client and server. In principle, she is also able to see the server's memory; however since the server does not see anything more of the client's queries than the adversary, the ability to see the server's memory does not reveal any information about the client's queries beyond what can already be inferred from the transcript alone.

- In the SSEReal world, the SSE scheme is executed honestly. The adversary observes the real transcript of each operation, and outputs a bit b.
- In the SSEIdeal world, the adversary sees a simulated transcript in place of the real transcript of the protocol. The simulated transcript is generated by a PPT algorithm S, known as the *simulator*, that has access to the leakage functions. For example, on $\mathsf{Setup}(\mathsf{DB})$, S returns a transcript from $S(\mathcal{L}^{\mathsf{Stp}}(\mathsf{DB}))$; and likewise for the Search and Update calls. The adversary eventually outputs a bit b.

These games are formally described in Appendix D. The scheme Σ is secure if the two worlds are indistinguishable.

Definition 3.1 (Adaptive security of SSE schemes)**.** *An SSE scheme Σ is $\mathcal{L}$-adaptively-secure, with respect to a leakage function $\mathcal{L}$, if for any polynomial-time adversary A issuing a polynomial number of queries $q(\lambda)$, there exists a PPT simulator S such that:*

$$\left|\mathbb{P}\left[\text{SSE}\textsc{Real}_A^{\Sigma}(\lambda, q) = 1\right] - \mathbb{P}\left[\text{SSE}\textsc{Ideal}_{A,S,\mathcal{L}}(\lambda, q) = 1\right]\right| = negl(\lambda).$$

3.4 Leakage Functions

In this section we define a few simple leakage functions. We begin with a common leakage function: the *search pattern* [CGKO06]. Most SSE schemes leak the fact that two search queries pertain to the same keyword. Indeed, unless some form of data-oblivious memory is used, when two searched keywords are equal, the search token will typically prompt the server to access the same sections of the encrypted database to retrieve the (same) document indices.

Formally, search pattern leakage is defined as follows. In its internal state, the leakage function records the list Q of every search query, in the form (u, w), where u is the *timestamp* (an index starting at 0 and increasing with every query) and w is the searched keyword. The search pattern is defined as a function $\mathbb{N} \to \mathcal{P}(\mathbb{N})$ with $\mathsf{sp}(w) = \{u : (u, w) \in Q\}$. Thus, sp leaks which search queries relate to the same keyword.

We also define the *history* $\mathsf{UpHist}(w)$ of each keyword w, following Bost [Bos16]. The function $\mathsf{UpHist}(w)$ outputs the list of all updates on keyword w: each element of the list is a tuple $(u, \mathsf{op}, \mathsf{ind})$ where u is the timestamp of the update, op is the operation, and ind is the updated index. For example, if there are two documents D_{ind_1} and D_{ind_2} matching w, such that D_{ind_1} was inserted at update 3, D_{ind_2} at update 7, and then D_{ind_2} was deleted at update 42, $\mathsf{UpHist}(w)$ will be $[(4, \mathsf{add}, \mathsf{ind}_1), (7, \mathsf{add}, \mathsf{ind}_2), (42, \mathsf{del}, \mathsf{ind}_2)]$.

4 Forward and Backward Privacy

Forward and backward privacy capture information leaked by a dynamic SSE scheme. At a high level, forward privacy considers privacy of the database and earlier search queries during updates, while backward privacy captures privacy of the database and updates to it during search queries. In this section, we formally define these privacy properties and present a generic transformation that meets these definitions, albeit at a cost. Our generic construction, and its instantiations with TWORAM [GMP16] and $\Sigma o\phi o\varsigma$ [Bos16], can be seen as a baseline solution that transforms any forward-private SSE scheme to provide backward privacy, at the cost of an additional roundtrip. We improve on the baseline solution in the following sections.

4.1 Forward Privacy

An SSE scheme is forward-private (or forward-secure) if Update queries do not leak which keywords are involved in the keyword/document pairs that are being updated. Forward privacy was informally defined in [SPS14]. Here we borrow the formal definition of [Bos16].

Definition 4.1 (Forward Privacy)**.** *A $\mathcal{L}$-adaptively-secure SSE scheme is* forward-private *iff the update leakage function $\mathcal{L}^{\mathsf{Updt}}$ can be written as:*

$$\mathcal{L}^{\mathsf{Updt}}(\mathsf{op}, \mathsf{in}) = \mathcal{L}'(\mathsf{op}, \{(\mathsf{ind}_i, \mu_i)\})$$

where the set $\{(\mathsf{ind}_i, \mu_i)\}$ captures all updated documents as the number of keywords μ_i modified in document ind_i; and $\mathcal{L}'$ is stateless.

If update queries are restricted to adding or deleting a single keyword/document pair, the scheme is forward-private iff we have $\mathcal{L}^{\mathsf{Updt}}(\mathsf{op}, w, \mathsf{ind}) = \mathcal{L}'(\mathsf{op}, \mathsf{ind})$. All forward-private schemes in this paper satisfy $\mathcal{L}^{\mathsf{Updt}}(\mathsf{op}, w, \mathsf{ind}) = \mathsf{op}$.

4.2 Backward Privacy

Backward privacy limits the information on the updates affecting keyword w that the server can learn upon a search query on w. Informally, an SSE scheme is backward-private (or backward-secure) if, whenever a keyword/document pair (w, ind) is added into the database and then deleted, subsequent Search queries on w do not reveal ind [SPS14]. Note that ind is revealed if a Search query is issued after (w, ind) is added, and before it is deleted.

Hence, we could argue that backward-private schemes are those whose search leakage is only a (stateless) function of $\mathsf{DB}(w)$, as this would only reveal information about document currently in the database (and not the deleted ones). However, this is not enough, as, even though the search leakage is reduced to $\mathsf{DB}(w)$, the update leakage could reveal the modified document/keyword pairs. A scheme with such leakage would reveal the indices of deleted documents, as the attacker could keep track of all the updated pairs, which is exactly what we want to prevent. As a consequence, in the security definitions, we must explicitly rule out such update leakage.

Moreover, obtaining a scheme with leakage that depends only on $\mathsf{DB}(w)$ would require hiding the pattern of updates as well as their number. Although hiding the former could be achieved, for example, using ORAM, this would result in expensive schemes. As a consequence, we define three flavors of backward privacy of decreasing strength, depending on how much metadata leaks about the inserted and deleted entries:

I. Backward privacy with insertion pattern:
leaks the documents *currently* matching w, when they were inserted, and the total number of updates on w.

II. Backward privacy with update pattern:
leaks the documents *currently* matching w, when they were inserted, and when all the updates on w happened (but not their content).

III. Weak backward privacy:
leaks the documents *currently* matching w, when they were inserted, when all the updates on w happened, and which deletion update canceled which insertion update.

Let us demonstrate the differences between these notions with an example. Consider the following sequence of updates, in the order of arrival: $(\mathsf{add}, \mathsf{ind}_1, \{w_1, w_2\})$, $(\mathsf{add}, \mathsf{ind}_2, \{w_1\})$, $(\mathsf{del}, \mathsf{ind}_1, \{w_1\})$, $(\mathsf{add}, \mathsf{ind}_3, \{w_2\})$. Let us consider the leakage for each definition after a search query on w_1. The first notion reveals ind_1 and that this entry was added at time 1. It also reveals that there were a total of 3 updates for w_1. The second notion, additionally reveals that updates on w_1 happened at time 1, 2, and 3. Finally, the third definition also reveals that the index that was added for w_1 at time 1 was removed at time 3.

In order to capture these notions, we introduce new leakage functions, starting with TimeDB. For a keyword w, $\mathsf{TimeDB}(w)$ is the list of all documents matching w, excluding the deleted ones, together with the timestamp of when they were inserted in the database. Formally, $\mathsf{TimeDB}(w)$ can be constructed from the query list Q as follows:

$$\mathsf{TimeDB}(w) = \{(u, \mathsf{ind}) \mid (u, \mathsf{add}, (w, \mathsf{ind})) \in Q \text{ and } \forall u',\ (u', \mathsf{del}, (w, \mathsf{ind})) \notin Q\}.$$

In particular $\mathsf{DB}(w) = \{\mathsf{ind} | \exists u \text{ s.t. } (u, \mathsf{ind}) \in \mathsf{TimeDB}(w)\}$. Note that TimeDB is completely oblivious to any document added to $\mathsf{DB}(w)$ that was later removed, but retains all other information. As such, TimeDB captures a strong notion of backward privacy revealing only the time of the insertion of the documents currently containing the search query w.

Then, we define $\mathsf{Updates}(w)$ which is the list of timestamps of updates on w. Formally,

$$\mathsf{Updates}(w) = \{u \mid (u, \mathsf{add}, (w, \mathsf{ind})) \text{ or } (u, \mathsf{del}, (w, \mathsf{ind})) \in Q\}.$$

$\mathsf{Updates}$ captures the leakage of the update pattern.

Finally, in order to capture the weakest notion of backward privacy, we use $\mathsf{DelHist}$. The *deletion history* $\mathsf{DelHist}(w)$ of w is the list of timestamps for all deletion operations, together with the timestamp of the inserted entry it removes. Formally, $\mathsf{DelHist}(w)$ is constructed as:

$$\mathsf{DelHist}(w) = \left\{(u^{\mathsf{add}}, u^{\mathsf{del}}) \mid \exists \mathsf{ind} \text{ s.t. } (u^{\mathsf{del}}, \mathsf{del}, (w, \mathsf{ind})) \in Q \text{ and } (u^{\mathsf{add}}, \mathsf{add}, (w, \mathsf{ind})) \in Q\right\}.$$

With these tools, we can formally define our three notions of backward privacy.

Definition 4.2 (Backward Privacy)**.** *A $\mathcal{L}$-adaptively-secure SSE scheme is* insertion pattern revealing backward-private *iff the search and update leakage functions $\mathcal{L}^{\mathsf{Srch}}$, $\mathcal{L}^{\mathsf{Updt}}$ can be written as:*

$$\begin{aligned}\mathcal{L}^{\mathsf{Updt}}(\mathsf{op}, w, \mathsf{ind}) &= \mathcal{L}'(\mathsf{op})\\ \mathcal{L}^{\mathsf{Srch}}(w) &= \mathcal{L}''(\mathsf{TimeDB}(w), a_w),\end{aligned}$$

where $\mathcal{L}'$ and $\mathcal{L}''$ are stateless.

A $\mathcal{L}$-adaptively-secure SSE scheme is update pattern revealing backward-private *iff the search and update leakage functions $\mathcal{L}^{\mathsf{Srch}}$, $\mathcal{L}^{\mathsf{Updt}}$ can be written as:*

$$\begin{aligned}\mathcal{L}^{\mathsf{Updt}}(\mathsf{op}, w, \mathsf{ind}) &= \mathcal{L}'(\mathsf{op}, w)\\ \mathcal{L}^{\mathsf{Srch}}(w) &= \mathcal{L}''(\mathsf{TimeDB}(w), \mathsf{Updates}(w)),\end{aligned}$$

where $\mathcal{L}'$ and $\mathcal{L}''$ are stateless.

A $\mathcal{L}$-adaptively-secure SSE scheme is weakly backward-private *iff the search and update leakage functions $\mathcal{L}^{\mathsf{Srch}}$, $\mathcal{L}^{\mathsf{Updt}}$ can be written as:*

$$\begin{aligned}\mathcal{L}^{\mathsf{Updt}}(\mathsf{op}, w, \mathsf{ind}) &= \mathcal{L}'(\mathsf{op}, w)\\ \mathcal{L}^{\mathsf{Srch}}(w) &= \mathcal{L}''(\mathsf{TimeDB}(w), \mathsf{DelHist}(w)),\end{aligned}$$

where $\mathcal{L}'$ and $\mathcal{L}''$ are stateless.

We can clearly see that backward privacy with insertion pattern implies update pattern revealing backward privacy, which itself implies weak backward privacy. Also observe that an insertion pattern revealing backward-private scheme has to be forward-private, and that if a scheme is both forward-private and weakly backward-private, then the leakage of update queries cannot depend on either the updated keyword (by definition of forward privacy) or the updated document index (by definition of weak backward privacy), so the leakage must be limited to the nature of the operation. This will indeed be the case for all schemes considered in this article.

Algorithm 1 Generic backward-private scheme $B(\Sigma)$ where Σ is an arbitrary SSE scheme and F is a PRF.

```
Setup(DB) :
1: $\Sigma.\mathsf{Setup}(\mathsf{DB})$, $K_\Sigma \xleftarrow{\$} \{0,1\}^\lambda$
Search($K_\Sigma, w, \sigma$; EDB)
1: Client and Server run $\Sigma.\mathsf{Search}(w)$, the client gets the list of results $R$.
   Client:
2: $K_w \leftarrow F(K_\Sigma, w)$
3: Decrypt $R$ as $(E_{K_w}(\mathsf{ind}_1, \mathsf{op}_1), \ldots, E_{K_w}(\mathsf{ind}_n, \mathsf{op}_n))$
4: Return $\{\mathsf{ind} : \exists i, (\mathsf{ind}_i, \mathsf{op}_i) = (\mathsf{ind}, \mathsf{add}) \wedge \forall j > i, (\mathsf{ind}_j, \mathsf{op}_j) \neq (\mathsf{ind}, \mathsf{del})\}$
Update($K_\Sigma$, add, $w$, ind, $\sigma$; EDB)
1: Client: $K_w \leftarrow F(K_\Sigma, w)$
2: Client and Server run $\Sigma.\mathsf{Update}(\mathsf{add}, w, E_{K_w}(\mathsf{ind}, \mathsf{op}))$
```

4.3 A Generic Two-Roundtrip Backward-Private Scheme

In this section, we show how to build a simple backward-private SSE scheme $B(\Sigma)$ starting from an arbitrary SSE scheme Σ. We start with a basic solution for clarity, then improve on it.

We alter Σ as follows. Instead of storing a document index ind, the client uploads a ciphertext $E_{K_w}(\mathsf{ind}, \mathsf{op})$, where E_{K_w} is a secret-key encryption scheme and $\mathsf{op} \in \{\mathsf{add}, \mathsf{del}\}$. The key K_w is specific to keyword w and is chosen by the client. The server sees only the resulting ciphertexts as K_w's are never revealed to it. The scheme Σ otherwise runs as normal. In particular, Search queries return the set of matching encrypted document indices $E_{K_w}(\mathsf{ind}, \mathsf{op})$. The client can then decrypt this set, remove deleted indices, and obtain the final set of document indices matching w.

A description of $B(\Sigma)$ is provided in Algorithm 1. Letting $\mathcal{I}$ denote the set of document indices, we assume $\mathcal{I} \times \{\mathsf{add}, \mathsf{del}\}$ embeds into the plaintext space of E_K, and we use the ciphertext space of E_K as the set of document indices for Σ. Note that Σ only needs to support add queries. The scheme $B(\Sigma)$ achieves update pattern revealing backward privacy, as Σ can leak any information about the modified keyword during updates, and some access pattern information during search. However, if Σ does not reveal any information about the past updates (*i.e.*, if Σ does not leak $\mathsf{UpHist}(w)$ but only $\mathsf{DB}(w)$), we can show that $B(\Sigma)$ guarantees backward-privacy with insertion pattern. Unfortunately, the only dynamic schemes which do not reveal $\mathsf{UpHist}(w)$ are based on ORAM, such as TWORAM [GMP16].

The $B(\Sigma)$ scheme, as described so far, has two drawbacks. The first drawback is that the server does not learn document indices in the clear and, hence, cannot return the matching documents. This is fine for a result-hiding scheme. However, a common use case of SSE schemes is to return actual documents, which are stored separately in an encrypted form. $B(\Sigma)$ can support this case with an additional roundtrip as follows. After the client computes the result of a search query, she sends document indices in the clear to the server. The server is then able to send the documents to the client. Hence, $B(\Sigma)$ is two-roundtrip protocol, assuming Σ requires a single roundtrip for its queries.

The second drawback of $B(\Sigma)$ is that deleted elements are never deleted on the server side. Moreover, since deleted elements are returned to the client on each search query, this also affects the communication cost and the amount of work necessary on the client side. We notice that this overhead can be avoided in the common scenario outlined above where the client sends cleartext document indices back to the server. In particular, it suffices for the client to send, together with the list of cleartext indices, an encryption of the same indices with a new key. Recall, that this list contains only the relevant indices with deleted elements removed by the client. Hence, the server can delete the old encrypted entries in the database and insert the updated ones. Essentially we are piggybacking a cleanup procedure on top of the Search protocol.

We denote a generic solution based on the above idea as $B'(\Sigma)$ and describe it in Algorithm 2. In $B'(\Sigma)$, the client keeps track of the number of times each keyword w has been queried in table T. Each time a search query is issued, results are re-encrypted using a fresh key derived from w and $\mathsf{T}[w]$. Keywords w in Σ are replaced by $w||\mathsf{T}[w]$, where $||$ denotes concatenation. In line 7 of the algorithm, re-encrypted indices are sent as Update queries for the sake of having a generic solution. However, typical SSE schemes would allow

Algorithm 2 Improved backward-private scheme $B'(\Sigma)$.

```
Setup(DB):
1: $\mathsf{T}[w] \leftarrow 0$ for all $w$, $K_\Sigma \xleftarrow{\$} \{0,1\}^\lambda$
2: $\mathsf{DB}' \leftarrow \mathsf{DB}$ where keywords $w$ are replaced by $w||0$
3: $\Sigma.\mathsf{Setup}(\mathsf{DB}')$
Search($K_\Sigma, w, \sigma; \mathsf{EDB}$)
1: Client: $K_w \leftarrow H(K_\Sigma, w, \mathsf{T}[w])$
2: Client and Server run $R \leftarrow \Sigma.\mathsf{Search}(w||\mathsf{T}[w])$    ▷ Server can erase all retrieved elements from memory
   Client:
3: Decrypt $R$ as $(E_{K_w}(\mathsf{ind}_1, \mathsf{op}_1), \ldots, E_{K_w}(\mathsf{ind}_n, \mathsf{op}_n))$
4: $R' \leftarrow \{\mathsf{ind} : \exists i, (\mathsf{ind}_i, \mathsf{op}_i) = (\mathsf{ind}, \mathsf{add}) \wedge \forall j > i, (\mathsf{ind}_j, \mathsf{op}_j) \neq (\mathsf{ind}, \mathsf{del})\}$
5: Send $R'$ to Server
6: $\mathsf{T}[w] \leftarrow \mathsf{T}[w] + 1$
7: for all $\mathsf{ind} \in R'$ do    ▷ In parallel
8:     Run $\mathsf{Update}(K_\Sigma, \mathsf{add}, w, \mathsf{ind}, \sigma; \mathsf{EDB})$
9: end for
Update($K_\Sigma, \mathsf{add}, w, \mathsf{ind}, \sigma; \mathsf{EDB}$)
1: Client: $K_w \leftarrow H(K_\Sigma, w, \mathsf{T}[w])$
2: Client and Server run $\Sigma.\mathsf{Update}(\mathsf{add}, w||\mathsf{T}[w], E_{K_w}(\mathsf{ind}, \mathsf{op}))$
```

all updates to be performed at once in a single roundtrip. We also expect that concrete choices of Σ may allow further optimisations. For example, directly using a result-hiding scheme for Σ would avoid having to encrypt the $(\mathsf{ind}, \mathsf{op})$ pairs before inserting them in Σ.

The scheme $B'(\Sigma)$ is intuitively backward-private since the server learns document indices only after the client has removed deleted indices. Moreover, since document indices are re-encrypted after each search, it achieves the notion of update pattern revealing backward privacy in the sense of Definition 4.2. We note that $B'(\Sigma)$ may achieve a stronger definition if one makes further assumptions on how updates are carried out in Σ. In particular, we name the B'(TWORAM) instantiation Moneta. Moneta achieves backward privacy with insertion pattern, but at a very high computational and communicational cost due to the use of TWORAM.

4.4 Fides: A Baseline Forward and Backward Private SSE Scheme

In this section, we briefly describe Fides, the instantiation of B' using $\Sigma o\phi o\varsigma$ [Bos16] (recall that $\Sigma o\phi o\varsigma$ is forward-private, but not backward-private). Fides guarantees forward privacy and update pattern revealing backward privacy. The former is due to the underlying SSE scheme, $\Sigma o\phi o\varsigma$, being forward-private, while the latter is the result of the B' construction. The formal statement on Fides' security is given by Theorem 1.

Theorem 1. *Define $\mathcal{L}_{\mathsf{Fides}}$ as:*

$$\mathcal{L}^{\mathsf{Updt}}_{\mathsf{Fides}}(\mathsf{op}, w, \mathsf{ind}) = \perp$$
$$\mathcal{L}^{\mathsf{Srch}}_{\mathsf{Fides}}(w) = (\mathsf{DB}(w), \mathsf{Updates}(w)).$$

Fides *is $\mathcal{L}_{\mathsf{Fides}}$-adaptively-secure.*

Let us analyze Fides' performance. Recall that $\Sigma o\phi o\varsigma$ is optimal for search and updates in terms of computation and communication. In contrast, Fides takes two rounds during search and has $O(a_w)$ computation and communication complexity, where a_w is the total number of update entries matching w. The cost of $O(a_w)$ is the worst case scenario since this cost can be amortized over all search queries for w. Similar to $\Sigma o\phi o\varsigma$, the updates in Fides are optimal (constant communication and computation).

Fides can be seen as a baseline for forward- and backward-private designs: it is simple to build, offers moderate computation overhead, and achieves a good level of security. In the next sections, we will propose

Algorithm 3 FS-RCPRF: Forward private SSE scheme from range-constrained PRF $\widetilde{F}$. H_1 and H_2 are hash functions.

```
Setup()
1: $K_\Sigma \xleftarrow{\$} \{0,1\}^\lambda$, $\mathbf{W}, \mathsf{EDB} \leftarrow$ empty map
2: return $(\mathsf{EDB}, K_\Sigma, \mathbf{W})$
```

```
Search($K_\Sigma, w, \sigma; \mathsf{EDB}$)
    Client:
1: $K_w||K'_w \leftarrow F_{K_\Sigma}(w)$, $c \leftarrow \mathbf{W}[w]$    ▷ $c = n_w - 1$
2: if $c = \perp$ then return $\emptyset$
3: $ST \leftarrow \widetilde{F}.Constrain(K_w, C_c)$    ▷ $C_c$ is the circuit evaluating to 1 on $\{0,\dots,c\}$
4: Send $(K'_w, ST, c)$ to the server.
    Server:
5: for $i = c$ to $0$ do
6:     $T_i \leftarrow \widetilde{F}(ST, i)$
7:     $UT_i \leftarrow H_1(K'_w, T_i)$
8:     $e \leftarrow \mathsf{EDB}[UT_i]$
9:     $\mathsf{ind} \leftarrow e \oplus H_2(K'_w, T_i)$
10:    Output each $\mathsf{ind}$
11: end for
```

```
Update($K_\Sigma, \mathsf{add}, w, \mathsf{ind}, \sigma; \mathsf{EDB}$)
    Client:
1: $K_w||K'_w \leftarrow F(K_\Sigma, w)$, $c \leftarrow \mathbf{W}[w]$
2: if $c = \perp$ then $c \leftarrow -1$
3: $T_w^{c+1} \leftarrow \widetilde{F}(K_w, c+1)$, $\mathbf{W}[w] \leftarrow c+1$
4: $UT_{c+1} \leftarrow H_1(K'_w, T_w^{c+1})$, $e \leftarrow \mathsf{ind} \oplus H_2(K'_w, T_w^{c+1})$
5: Send $(UT_{c+1}, e)$ to the server.
    Server:
6: $\mathsf{EDB}[UT_{c+1}] \leftarrow e$
```

schemes that avoid inefficiencies such as the additional roundtrip and the high communication overhead at the cost of being only weakly backward-private.

5 Diana: Forward-Secure SSE with Very Low Overhead

In this section, we describe a generic way to construct forward-private searchable encryption from constrained PRFs on $\mathbb{N}$ with respect to the range family of circuits $\mathcal{C} = \{C_c | C_c(x) = 1 \Leftrightarrow 0 \le x \le c\}$. We will see that $\Sigma o\phi o\varsigma$ [Bos16] can be seen as an instantiation of this scheme, and then provide a much more efficient one based on the GGM PRF [GGM84], which we call Diana.

5.1 FS-RCPRF: Forward-Secure SSE from Range Constrained PRFs

Let $\widetilde{F} : \{0,1\}^\lambda \times \{0,\dots,n_{\max}\} \to \{0,1\}^\lambda$ be a constrained PRF with respect to the class of range circuits $\mathcal{C}$ defined above. Also, let F be a 2λ-bit PRF. Algorithm 3 describes FS-RCPRF, a forward-secure scheme based on the range-constrained PRF $\widetilde{F}$. The simple idea behind FS-RCPRF is that update tokens for entries matching keyword w are generated using $\widetilde{F}$ in counter mode, where the counter is incremented every time a new entry matching w is inserted. Then, during search, the client gives to the server the constrained key allowing only the evaluation of $\widetilde{F}$ on $\{0,\dots,n_w\}$. The resulting scheme can be seen as a generalization of the dynamic scheme of Cash *et al.* [CJJ$^+$14], where during the search the client gives to the server the constrained key of w instead of the master key K_w.

The intuition for the security of FS-RCPRF is simple: as the adversary only gets to see the CPRF keys during searches for ranges corresponding to already inserted entries, she cannot predict the evaluation of the PRF for inputs outside of these ranges, and in particular for newly inserted entries. Hence updates leak no information. Theorem 2 states the formal security of FS-RCPRF. Its proof is deferred to Appendix A.

Theorem 2 (Adaptive security of FS-RCPRF)**.** *Define* $\mathcal{L}_{FS} = (\mathcal{L}_{FS}^{\mathsf{Srch}}, \mathcal{L}_{FS}^{\mathsf{Updt}})$ *as:*

$$\mathcal{L}_{FS}^{\mathsf{Srch}}(w) = (\mathsf{sp}(w), \mathsf{UpHist}(w))$$
$$\mathcal{L}_{FS}^{\mathsf{Updt}}(\mathsf{add}, w, \mathsf{ind}) = \perp.$$

FS-RCPRF is $\mathcal{L}_{FS}$*-adaptively-secure.*

The adaptive security of FS-RCPRF is shown in the random oracle model (ROM), but the ROM is not needed for non-adaptive security.

Reinterpreting $\Sigma o\phi o\varsigma$ with constrained PRFs The $\Sigma o\phi o\varsigma$ construction is based on the iteration of a trapdoor permutation (TDP) π to generate the update tokens in a way that prevents the server from predicting them. $\Sigma o\phi o\varsigma$ can be reinterpreted using our framework by constructing a TDP-based range-constrained PRF $\widetilde{F}_\Sigma$ (in the following paragraph, we re-use the notation of [Bos16], which overrules our own).

The master key $\widetilde{F}_\Sigma$ is composed of an RSA key SK and an element $ST_0 \in \mathbb{Z}_N$ where each can be pseudo-randomly generated from a random λ-bit key. $\widetilde{F}((\mathsf{SK}, ST_0), c) = \pi_{\mathsf{SK}}^{-c}(ST_0)$ where π^{-c} is the c-fold iteration of π^{-1}. The constrain algorithm will then be the following (we identify the circuit constraining to the range $\{0, \dots, n\}$ with the integer n):

$$\widetilde{F}.Constrain((\mathsf{SK}, ST_0), n) = (\mathsf{PK}, \pi_{\mathsf{SK}}^{-n}(ST_0), n) = (\mathsf{PK}, ST_n, n).$$

Finally, the constrained evaluation function is

$$\widetilde{F}.Eval((\mathsf{PK}, ST_n, n), c) = \pi_{\mathsf{PK}}^{n-c}(ST_c).$$

We can easily reduce the constrained-PRF security of $\widetilde{F}$ to the hardness of the RSA assumption, and directly deduce the security of $\Sigma o\phi o\varsigma$ from Theorem 2.

5.2 Diana, a GGM instantiation of FS-RCPRF

In this section we present a range-constrained PRF and then use it to instantiate FS-RCPRF.

We can easily construct a simple and efficient range-constrained PRF from the tree-based GGM PRF [GGM84]. This instantiation has been described by Kiayias *et al.* [KPTZ13] and is called best range cover (BRC).

Let $G : \{0,1\}^\lambda \to \{0,1\}^{2\lambda}$ be a pseudo-random generator (PRG), $G_0(k)$ and $G_1(k)$ be the first and second half of $G(k)$. The GGM PRF on n-bit integers is defined as $F_K(x) = G_{x_n-1}(\dots(G_{x_1}(G_{x_0}(K))))$ where $x_{n-1} \dots x_0$ is the binary representation of x. The leaves of the tree are the output values of F, and they can be labeled according to the corresponding input, and the partial evaluation of F (*i.e.* the iterated evaluation of G, but only on the first $\ell < n$ bits) are the inner nodes of the tree.

To constrain F to the input range $[0, c-1]$, we generate the nodes of the tree covering exactly the leaves with labels in $[0, c-1]$. In practice if the binary representation of c is $c_{n-1} \dots c_0$, the punctured key would be $\left\{G_0(G_{c_{i-1}}(\dots(G_{c_0}(K))))\right\}$ for i such that $c_i = 1$.

We use the above range CPRF to instantiate FS-RCPRF and call this instantiation Diana. Note that, Diana is almost identical to the ARX-EQ scheme [PBP16]. However ARX-EQ was not formally proven, and FS-RCPRF provides a more general framework on how to construct forward-private SSE schemes.

Let us analyze the efficiency of Diana. Updates need $O(\log n_{\max})$ computation, where $n_{\max}$ is the maximum number of entries matching a keyword: CPRF computes a tree's leaf from its root. Similarly, during search, the server has to compute all the leaves of the tree within a given range. This can be done efficiently in $O(n_w)$ calls to the PRG, where n_w is the number of matches on search keyword w: there are $O(n_w)$ tree nodes to compute in total and each node can be generated using a single PRG call. In terms of communication complexity, Diana is optimal for updates, and sends $O(\log n_w)$ tree nodes during a search query.

In theory, this is worse than $\Sigma o\phi o\varsigma$' optimal computational and communication complexity, but, as we will see in Section 7.1, Diana uses symmetric primitives that are much faster than $\Sigma o\phi o\varsigma$' RSA. Also, since nodes in the tree will be 128-bit keys, we can set $n_{\max}$ to 2^{32} and still have search tokens only twice as big as $\Sigma o\phi o\varsigma$' 2048-bit tokens.

5.3 $\mathsf{Diana}_{\mathsf{del}}$: Backward-Secure SSE from Range-Constrained and Puncturable PRFs

The FS-RCPRF construction, and its instantiation Diana, do not support deletions. Schemes of this type can be extended to support deletions by letting the client and the server maintain two instances of the construction, one for insertions and one for deletions. Then, during a search query, the server can compute the difference between the two result sets to compute the list of documents matching the query (i.e., without the deleted entries). This solution, however, is not backward-private as the server trivially learns the deleted entries. To this end, we propose FS-RCPRF$_{\mathsf{del}}$, which also uses two SE instances but exploits constrained PRFs to guarantee weak backward privacy.

The key idea behind FS-RCPRF$_{\mathsf{del}}$ is to extend the set of constraints supported by the underlying constrained PRF used in Section 5.1. In order to support backward privacy, we make use of constrained PRF $\tilde{F}$ that is not only range-constrained (for forward privacy) but is *also* punctured on the deleted entries (for backward privacy). Hence, the constrained key of $\tilde{F}$ enforces the predicate $C_{c,x_1,\dots,x_n}(x) = 1$ iff $x \in [0,c]$ and $\forall i, x \neq x_i$. The values $x_1, \dots, x_n$ correspond to deleted entries that the server should not learn. Unfortunately, a naive implementation of $\tilde{F}$ requires the client to store all deleted entries $x_1, \dots, x_n$ since the order of deletions and insertions can be arbitrary. Our construction avoids this storage overhead on the client's side by letting the server store the deleted entries in an encrypted form.

We now combine the above ideas and describe FS-RCPRF$_{\mathsf{del}}$. The client and the server maintain two forward-private SE instances: one for insertions and one for deletions. Every time the client wants to insert (w, ind) with the counter c, it proceeds as in FS-RCPRF and inserts the pair in the first SE instance, as in Algorithm 3. In addition, it also pushes the pair $(F'(K_w, (w, \mathsf{ind})), Enc_{K'}(c))$, where F' and Enc are a PRF and a CPA encryption scheme, to the server who stores these pairs in a map. In order to delete (w, ind), the client inserts the entry $(w, F'(K_w, (w, \mathsf{ind})))$ in the second SE instance. Then, during search query for w, the client proceeds as follows. It requests a search for w on the second SE instance (i.e., the one that stores deleted entries). As a result, the server gets the associated tags $F'(K_w, (w, \mathsf{ind}))$ for the deleted entries, uses them to retrieve encrypted x_i's from the map, and sends them back to the client. The client then constrains the PRF using x_i's and uses it to run a search on the first SE instance. Note that this solution assumes that the same index ind is never reused: once the entry (w, ind) has been deleted, it can no longer be re-added.

The above solution is not ideal as it requires an additional roundtrip with large communication from the server to the client. Also, it can only guarantee weak backward privacy, as the server learns when the deletions occurred.

Similar to FS-RCPRF, we instantiate FS-RCPRF$_{\mathsf{del}}$ with the GGM PRF and call the resulting scheme $\mathsf{Diana}_{\mathsf{del}}$. The constrained key, instead of consisting of the covering nodes of the full range as in Section 5.2, will be constructed as the set of nodes covering the ranges $[0, x_1 - 1], [x_1 + 1, x_2 - 1], \dots, [x_n + 1, c]$ (assuming that x_i's are in increasing order). This approach will result in large keys when the number of deletions is large: the number of tree nodes to be sent will be in the order of $d_w \cdot \log(n_w/d_w)$. (assuming uniformly distributed deletions).

6 Janus: Weak Backward Security from Puncturable Encryption

The solutions presented in Section 5.3 suffer from high inefficiencies, by requiring either client storage linear in the number of deletions, or multiple roundtrips with high communication complexity. In this section, we show how to achieve (weak) backward security in a single roundtrip, using puncturable encryption with incremental punctures.

6.1 Puncturable Encryption

A puncturable encryption (PE) scheme is a public-key encryption scheme that allows to *puncture* the secret key to prevent the decryption of some messages. More precisely, for such schemes, the plaintexts are encrypted and attached to a *tag*, and the secret key is punctured on a set of tags so that decryption of ciphertexts attached to those tags is impossible. Puncturable encryption has been introduced by Green and Miers as a way to achieve forward security in an asynchronous setting [GM15]. We adopt the same formalism and definitions, except we fix the number of tags per message to 1.

A puncturable encryption scheme PPKE with message space $\mathcal{M}$ and tag space $\mathcal{T}$ is a triple of algorithms $(\mathsf{KeyGen}, \mathsf{Encrypt}, \mathsf{Puncture}, \mathsf{Decrypt})$ with the following syntax:

- $\mathsf{KeyGen}(1^\lambda)$ outputs a public key PK and an initial secret key SK_0.
- $\mathsf{Encrypt}(PK, M, t)$ outputs the encryption CT of $M \in \mathcal{M}$ attached to the tag $t \in \mathcal{T}$.
- $\mathsf{Puncture}(\mathsf{SK}_i, t)$ outputs a new secret key SK_{i+1} able to decrypt any ciphertext SK_i can decrypt, except for ciphertexts encrypted with the tag t.
- $\mathsf{Decrypt}(\mathsf{SK}_i, CT, t)$ outputs a plaintext M or $\perp$ if the decryption fails.

Correctness is achieved if a plaintext M encrypted with tag t decrypts back to M when using the secret key punctured on any set of tags that does not contain t.

The $\mathsf{IND\text{-}PUN\text{-}ATK}$ security games – with $\mathsf{ATK} \in \{\mathsf{CPA}, \mathsf{CCA}\}$ – capture the security of puncturable encryption. We recall the $\mathsf{IND\text{-}PUN\text{-}CPA}$ game (we will not use CCA security in this work) in a simplified version.

Definition 6.1 (Security of puncturable encryption)**.** *Let* PPKE *be a puncturable encryption scheme. The game* $\mathsf{IND\text{-}PUN\text{-}CPA}_{\mathsf{PPKE},A}$ *with adversary* A *is defined using three phases as follows:*

Setup Phase *The challenger initializes two empty sets* P, C, T*, a counter* n *to 0, and runs* $(\mathsf{PK}, \mathsf{SK}_0) \leftarrow \mathsf{PPKE.KeyGen}(1^\lambda)$*. Finally, he randomly picks* $b \xleftarrow{\$} \{0,1\}$*.*

Query Phase *The adversary can adaptively issue the following queries:*

Puncture(t) *The challenger increments* n*, computes* $\mathsf{SK}_n \leftarrow \mathsf{PPKE.Puncture}(\mathsf{SK_{n-1}}, \mathsf{t})$ *and adds* t *to* P*.*

Corrupt$()$ *The first time the adversary issues this query, if* $T \subseteq P$*, the challenger returns the most recent secret key* SK_n*, and sets* $C \leftarrow P$*. All subsequent queries return* $\perp$*.*

Challenge(M_0, M_1, t) *If the adversary previously issued a Corrupt query and* $t \notin C$*, the challenger rejects the challenge. Otherwise, the challenger returns* $CT \leftarrow \mathsf{PPKE.Encrypt}(\mathsf{PK}, M_b, t)$ *to the adversary and adds* t *to* T*.*

Guess Phase *The adversary outputs a guess* b' *of* b*.*

The game ensures that the adversary can get challenge ciphertexts only for tags on which the secret key has been punctured.

We say that PPKE *is* $\mathsf{IND\text{-}PUN\text{-}CPA}$ *secure if for all polynomial-time adversaries* A*:*

$$\begin{aligned}\mathbf{Adv}^{\mathsf{pun-cpa}}_{\mathsf{PPKE},A}(\lambda) =& |\mathbb{P}[\mathsf{IND\text{-}PUN\text{-}CPA}_{\mathsf{PPKE},A}(\lambda) = 1] \\ &\quad - \mathbb{P}[\mathsf{IND\text{-}PUN\text{-}CPA}_{\mathsf{PPKE},A}(\lambda) = 0]| \\ \leq& \ negl(\lambda).\end{aligned}$$

In the Janus construction, described in Section 6.3, we will encrypt the document indices using puncturable encryption, with tags that are pseudo-randomly generated from the document-keyword pairs. There will be a different key for each keyword, and when we want to delete an entry for a specific keyword, we will puncture the associated key on the tag derived from the document-keyword pair. Upon a search query, the client will give to the server the associated punctured secret key, with which he will only be able to decrypt non-deleted entries.

In this paper, we will use the Green-Miers puncturable encryption scheme [GM15], described in Appendix C.

6.2 Incremental Puncture

The punctured keys will (often) grow with the number of punctures (or be very large), and it will be impractical to store them on the client side. To avoid this issue, we use an additional feature of the Green-Miers scheme, which we call incremental puncture.

In our setting, we will see that it is very handy to be able to express the Puncture algorithm as a function of a constant-sized fraction of the secret key. The secret key of the Green-Miers puncturable encryption scheme is, after n punctures, $\mathsf{SK}_n = (sk_0, sk_1, \ldots, sk_n)$, and the puncture algorithm is such that

$$\begin{aligned}\mathsf{Puncture}(\mathsf{SK}_n, t) &= (sk_0', sk_1, \ldots, sk_n, sk_{n+1})\\ \text{where } (sk_0', sk_{n+1}) &= \mathsf{IncPuncture}(sk_0, t).\end{aligned}$$

By using this PE scheme, the client will only have to store the sk_0 part of the secret key, and outsource the rest to the server. The client's storage will stay linear in the number of keywords, and most of the storage burden will still be born by the server.

6.3 The Janus Construction

Janus, similar to the constructions in Section 5.3, uses two forward-secure searchable encryption instances: Σ_{add} to store the newly inserted indices encrypted with the puncturable encryption scheme (the insertion instance), and Σ_{del} to store the punctured key elements (the deletion instance). There is a different encryption key for each keyword and the client stores the sk_0 part of each key locally. During the search for w, the client sends the associated key part and runs the search protocol of the SE scheme for both instances. As a result, the server obtains the encrypted indices from the insertion instance and all the remaining key parts from the deletion instance. She will then be able to decrypt all the non-deleted (*i.e.* not punctured) indices.

Still, there is an important problem to tackle: once the secret key for w has been revealed to the server, it can no longer be used by the client to encrypt the index of the documents matching w that will be inserted in the future. As a consequence, we need to change the encryption key after every search. Yet, we do not need to re-encrypt the already revealed indices (*a.k.a.* the result indices) with the new key: the adversary already learned them, and, as the Σ_{add} and Σ_{del} schemes used in practice will leak the search pattern, she can keep track of the results over repeating search queries.

So, in the first version of our construction, the server will explicitly keep the results in a cache. This cache is also interesting from a performance point of view: each matching index will be decrypted at most once, and all the results from previous searches on a given keyword can be stored close to each other, increasing storage locality.

Description of Janus Janus is described in Algorithm 4. It uses two response-revealing (insertion-only) dynamic SSE schemes Σ_{add} and Σ_{del}. Σ_{add} and Σ_{del} might be different for efficiency or security purposes, but in the proof, we will assume that they are forward-secure. Janus also uses a PRF F and an incremental puncturable encryption scheme PPKE.

The client stores locally a table containing for each keyword w the initial key share $sk_0[w]$ of a PE (WLOG we can assume that this key share contains the public key). To insert a new entry (w, ind), the client encrypts it with the PE scheme with the key $sk_0[w]$, using a pseudo-random value $F(w, \mathsf{ind})$ as a tag. He then inserts this ciphertext as a new entry matching w in Σ_{add}. To delete the entry (w, ind), the client computes the tag $t = F(w, \mathsf{ind})$ and (incrementally) punctures $sk_0[w]$ on this tag. He then updates the initial key share of w and pushes the new key element sk_t to the server by inserting the entry (w, sk_t) in Σ_{del}. Finally, to search, the client runs a search on w for both Σ_{add} and Σ_{del}. The server now has access to the ciphertexts encrypting the inserted indices and to the key elements necessary to decrypt them. Note that she will only be able to decrypt the ciphertexts for which the key has not been punctured, *i.e.* the non deleted entries.

After a search query on w, the same encryption key cannot be used to encrypt new entries matching w: the server can reuse the old key to decrypt even the newly deleted entries since the key would not have been

Algorithm 4 Janus: weakly backward-secure SSE.

```
Setup()
1: $(\mathsf{EDB}_{\mathsf{add}}, K_{\mathsf{add}}, \sigma_{\mathsf{add}}) \leftarrow \Sigma_{\mathsf{add}}.\mathsf{Setup}()$
2: $(\mathsf{EDB}_{\mathsf{del}}, K_{\mathsf{del}}, \sigma_{\mathsf{del}}) \leftarrow \Sigma_{\mathsf{del}}.\mathsf{Setup}()$
3: $K_{tag}, K_S \leftarrow \{0,1\}^\lambda$, $\mathbf{PSK}, \mathbf{SC}, \mathsf{EDB}_{cache} \leftarrow$ empty map
4: return $((\mathsf{EDB}_{\mathsf{add}}, \mathsf{EDB}_{\mathsf{del}}, \mathsf{EDB}_{cache}), (K_{\mathsf{add}}, K_{\mathsf{del}}, K_{tag}, K_S), (\sigma_{\mathsf{add}}, \sigma_{\mathsf{del}}, \mathbf{PSK}, \mathbf{SC}))$
```

```
Search($K_\Sigma, w, \sigma$; EDB)
    Client:
 1: $i \leftarrow \mathbf{SC}[w]$.
 2: if $i = \perp$ return $\emptyset$
 3: Send $sk_0 = \mathbf{PSK}[w]$ to the server.
 4: $\mathbf{PSK}[w] \leftarrow \mathsf{PPKE.KeyGen}(1^\lambda)$, $\mathbf{SC}[w] \leftarrow i + 1$.
 5: Send $\mathsf{tkn} \leftarrow F(K_S, w)$ to the server.
    Client (C) & Server (S):
 6: C and S run $\Sigma_{\mathsf{add}}.\mathsf{Search}(K_{\mathsf{add}}, w||i, \sigma_{\mathsf{add}}; \mathsf{EDB}_{\mathsf{add}})$.
    The server gets a list $((ct_1, t_1^{\mathsf{add}}), \ldots, (ct_n, t_n^{\mathsf{add}})$ of
    ciphertexts and tags.
 7: C and S run $\Sigma_{\mathsf{del}}.\mathsf{Search}(K_{\mathsf{del}}, w||i, \sigma_{\mathsf{del}}; \mathsf{EDB}_{\mathsf{del}})$.
    The server gets a list $((sk_1, t_1^{\mathsf{del}}), \ldots, (sk_m, t_m^{\mathsf{del}}))$
    of key elements.
 8: S decrypts the ciphertexts with $\mathsf{SK} = (sk_0, sk_1, \ldots, sk_m)$, and obtains the list
    $NewInd = ((\mathsf{ind}_1, t_1), \ldots, (\mathsf{ind}_\ell, t_\ell))$.
    Server:
 9: $OldInd \leftarrow \mathsf{EDB}_{cache}[\mathsf{tkn}]$
10: Remove from $OldInd$ the indices whose tags are
    in $\{t_j^{\mathsf{del}}\}$.
11: $Res \leftarrow OldInd \cup NewInd$, $\mathsf{EDB}_{cache}[\mathsf{tkn}] \leftarrow Res$
12: return $Res$
```

```
Update($K_\Sigma$, add, $w$, ind, $\sigma$; EDB)
 1: $t \leftarrow F_{K_{tag}}(w, \mathsf{ind})$
 2: $sk_0 \leftarrow \mathbf{PSK}[w]$, $i \leftarrow \mathbf{SC}[w]$
 3: if $sk_0 = \perp$ then
 4:     $sk_0 \leftarrow \mathsf{PPKE.KeyGen}(1^\lambda)$, $\mathbf{PSK}[w] \leftarrow sk_0$
 5:     $i \leftarrow 0$, $\mathbf{SC}[w] \leftarrow i$
 6: end if
 7: if op = add then
 8:     $ct \leftarrow \mathsf{PPKE.Encrypt}(sk_0, \mathsf{ind}, t)$
 9:     Run $\Sigma_{\mathsf{add}}.\mathsf{Update}(K_{\mathsf{add}}, \mathsf{add}, w||i, (ct, t), \sigma_{\mathsf{add}}; \mathsf{EDB}_{\mathsf{add}})$
10: else                                          ▷ op = del
11:     $(sk_0', sk_t) \leftarrow \mathsf{PPKE.IncPuncture}(sk_0, t)$
12:     Run $\Sigma_{\mathsf{del}}.\mathsf{Update}(K_{\mathsf{del}}, \mathsf{add}, w||i, (sk_t, t), \sigma_{\mathsf{del}}; \mathsf{EDB}_{\mathsf{del}})$
13:     $\mathbf{PSK}[w] \leftarrow sk_0'$
14: end if
```

punctured on the corresponding tags. Janus avoids this by requiring the client to generate a new key for w after a search and encrypt new entries of w using this key. As discussed earlier, the server can keep the results of previous search queries and retrieve them the next time w is searched. This does not affect the security of the scheme since the server has already learnt earlier search results on w.

Security of Janus Janus is a forward-private and weakly backward-private SSE scheme. The former comes directly from the forward security of Σ_{add} and Σ_{del}. Let us consider backward security. The server has access to the decryption key of w's entries only during the search query for w. Moreover, this key allows her to decrypt only the entries that have been added since the last search for w and have not yet been deleted. Hence, the deleted indices remain hidden. Note that weak backward security is the strongest definition we can achieve with Janus as the server can determine which of the inserted queries were later deleted as well as the timestamps of these events. Also note that Janus does not allow re-insertion of document/keyword pairs that were previously deleted.

The formal security claim is given in Theorem 3, and its proof is postponed to Appendix B.

Theorem 3 (Adaptive Security of Janus)**.** *If Σ_{add} and Σ_{del} are two $\mathcal{L}_{FS}$-adaptively-secure SSE schemes, PPKE is IND-PUN-CPA secure, and F is a PRF, then Janus is $\mathcal{L}_{wBS}$-adaptively secure, with $\mathcal{L}_{wBS} = (\mathcal{L}_{wBS}^{\mathsf{Srch}}, \mathcal{L}_{wBS}^{\mathsf{Updt}})$ defined as*

$$\mathcal{L}_{wBS}^{\mathsf{Srch}}(w) = (\mathsf{sp}(w), \mathsf{TimeDB}(w), \mathsf{DelHist}(w))$$
$$\mathcal{L}_{wBS}^{\mathsf{Updt}}(\mathsf{op}, w, \mathsf{ind}) = \mathsf{op}.$$

Note that in this theorem, $\mathcal{L}_{FS}$ specifically refers to the leakage of a forward-secure scheme as defined in Theorem 2.

Efficiency The computational and communication complexity of Janus can easily be derived from Σ_{add} and Σ_{del}. In particular, it has the same complexity for insertion (resp. deletion) updates as Σ_{add} (resp. Σ_{del}). To analyze search queries, let $T_{\mathsf{add}}(n)$ and $T_{\mathsf{del}}(n)$ be the computational complexities of the search protocols of Σ_{add} and Σ_{del}, respectively, where n is the size of a result set. Then, Janus' search complexity for a keyword w with a_w insertions, d_w deletions, and $n_w = a_w - d_w$ non-deleted matching results, is $T_{\mathsf{add}}(a_w) + T_{\mathsf{del}}(d_w) + O(n_w \cdot d_w)$. The last term comes from the fact that a decryption of the PE scheme has complexity linear in the number of punctures. When instantiated with Diana or $\Sigma o\phi o\varsigma$, Janus thus has search complexity $O(a_w + d_w + n_w \cdot d_w) = O(n_w \cdot d_w)$.

In terms of communication for search queries, Janus also inherits from the complexity of Σ_{add}, and Σ_{del}. Let $C_{\mathsf{add}}(n)$ and $C_{\mathsf{del}}(n)$ be the communication complexities of search protocols of Σ_{add} and Σ_{del}, respectively, for a keyword that was inserted n times. Then, the communication complexity $C_{\mathsf{Janus}}(a_w, d_w)$ for a keyword that was inserted a_w times and deleted d_w times is $C_{\mathsf{add}}(a_w) + C_{\mathsf{del}}(d_w)$. Also, the number of roundtrips is the maximum number of roundtrips between Σ_{add} and Σ_{del}. Hence, when instantiated either with $\Sigma o\phi o\varsigma$ or Diana, Janus has single-roundtrip search and updates protocols. In the case of $\Sigma o\phi o\varsigma$, the search communication complexity is optimal (constant), and for Diana, it is $O(\log(a_w) + \log(d_w))$.

6.4 Reducing Storage Overhead

In practice, the storage overhead of Janus is quite high: the client needs to store 3 group elements (at least 256 bits each) for every keyword, while each ciphertext on the server side consists of the masked index, two group elements and the tag, and 3 group elements and a tag for each key share. To reduce the overhead at the client, we use a trick similar to the one used in $\Sigma o\phi o\varsigma$: we pseudo-randomly generate the encryption scheme's parameters and key elements sk_i from a master key and the number of punctures done on the secret key. The client does not need to store the public key as he can directly encrypt the plaintext indices from the scheme's parameters (and this will actually be faster). As a result, the client has to store only the number of deleted entries for each w, which he does already if Σ_{del} is instantiated with Diana. This modification is described in detail in Algorithm 11 (Appendix C).

A similar trick can be used to reduce the storage on the server side. Indeed, one of the three group elements stored for each entry is a random blinding element, which can be generated pseudo-randomly using a PRF applied on the keyword/document pair (w, ind) to be encrypted. As the blinding element is part of the ciphertext, and as it is now a (deterministic) function of the pair (w, ind), the tag is now redundant an can be omitted. This modification is also described in Algorithm 11.

6.5 Security of Janus Against Weaker Adversaries

We showed that Janus protects against persistent adversaries (*e.g.*, a malicious server) and guarantees both forward and backward privacy. In this section, we analyze its resistance against weaker adversaries. First, we consider a *snapshot adversary* who is able to see the encrypted database at one (or more) instant – *e.g.* in case of a disk theft or subpoena. Then, we consider the security of Janus against a *late-persistent* adversary that obtains control over the server sometime after the client has outsourced his data and, possibly executed some queries — *e.g.* in case of malware.

Janus, as is, does not protect against a snapshot adversary since the cached results are kept in plaintext on the server side. Beside trivially revealing the cached content, this can also lead to the recovery of some of the queries. This, in turn, can be used for leakage-abuse attacks in the manner of file injections attacks [ZKP16] adapted to using a single (or multiple) snapshot of EDB (and in particular of EDB_{cache}).

To fix this problem, we propose to encrypt EDB_{cache} using a key that is not permanently stored at the server and maintain EDB_{cache} in a *history-independent (HI) data structure* as follows. In particular, the content of EDB_{cache} relevant to w is encrypted using a keyword-specific symmetric key K_w. To this end, we modify line 5 of Algorithm 4 to $\mathsf{tkn}||K_w \leftarrow F(K_S, w)$ where the client also sends K_w to the server. Then

Table 2 – Size of the databases used in the evaluation, and the amount of storage needed for $\mathbf{W}$ and EDB

K	N	$\mathbf{W}$	EDB
$222 \cdot 10^3$	$1.9 \cdot 10^7$	4.6 MB	615 MB
$2.68 \cdot 10^6$	$1.9 \cdot 10^8$	46 MB	6.3 GB
$21.8 \cdot 10^6$	$1.9 \cdot 10^9$	365 MB	47 GB
$42.9 \cdot 10^6$	$3.8 \cdot 10^9$	720 MB	95 GB

the server uses K_w to decrypt and re-encrypt EDB_{cache} as needed, using an IND-CPA-secure secret-key encryption scheme E_{K_w}. Once the Search query is processed, the server discards K_w; in particular it must not be stored in EDB.

Unfortunately, encryption alone is not sufficient as the implementation of EDB_{cache} could leak additional information, such as the time of insertion/modification of data, or the size of previous, now discarded, values. To this end, we rely on *history-independent (HI) data structures* [NT01] whose goal is to hide exactly this kind of side-channel information. Note that if Σ_{add} and Σ_{del} are instantiated with existing forward-secure schemes (SPS [SPS14], $\Sigma o\phi o\varsigma$ [Bos16], or Diana), history-independence is not an issue as the snapshot adversary learns at most the update leakage, reduced to the list (op_i) with $\mathsf{op}_i = \mathsf{add}$ if the i-th update was an insertion, and $\mathsf{op}_i = \mathsf{del}$ otherwise. Though HI data structures come with an additional overhead, the state-of-the-art constructions are practical [BS13].

The security of the above approach relies on cooperation from the server who is required to use encryption and HI structures for EDB_{cache} and erase K_w from memory once he finishes en/de-crypting EDB_{cache}. Note that snapshot attacks are essentially attacks against the server, more so than against the client: we are protecting from the attacker information learned by the server.

Despite the assumptions we have just outlined, it is clear that storing the cache in encrypted form is a vast improvement over storing this information in cleartext. It is also a cheap solution: symmetric encryption is extremely fast on modern processors, especially in the presence of specialized instructions such as AES-NI. Encrypting EDB_{cache} would not significantly impact performance, relative to the decryption of punctured encryption schemes, or running the two SSE schemes Σ_{add} and Σ_{del}.

Let us now consider Janus against *late-persistent* adversaries. In this case, we strive to obtain the following backward privacy: even if a deleted entry matched a search query processed *before* the corruption, it should be infeasible for the adversary to recover the associated document index. Symmetrically encrypting EDB_{cache}, as in the case of the snapshot adversary, is no longer sufficient as the encryption key K_w will eventually be revealed. Instead, we require that the server encrypt results with the PE scheme, using the public key for the newly generated secret key (line 4 in Algorithm 4).

7 Performance Evaluation

We implemented and evaluated some of the schemes presented in this paper. The PRF has been instantiated with HMAC, and we chose Blake2b as the underlying hash function. For the GGM range-constrained PRF $\widetilde{F}$, we used AES in counter mode for the pseudo-random generator G. The keyed hash function H used in Diana is the AES block cipher used in Matyas-Meyer-Oseas mode [PGV94].

The code was written in C/C++, with many of the symmetric cryptography function implemented in assembly, in particular using AES-NI for the AES-based primitives. The code is OpenSource and freely accessible [Bos17].

We ran our experiments on a desktop computer with a single Intel Core i7 4790K 4.00GHz CPU (with 8 logical cores), 16GB of RAM, and a 250 GB Samsung 850 EVO SSD dedicated to the experiment. The key-value store is implemented with RocksDB [Fac].

7.1 Performance of Diana

Our evaluation of Diana uses 4 different, synthetically generated, data sets, each of different size. A quick summary of the statistics of the data sets, the size of the resulting encrypted databases, and the size of the

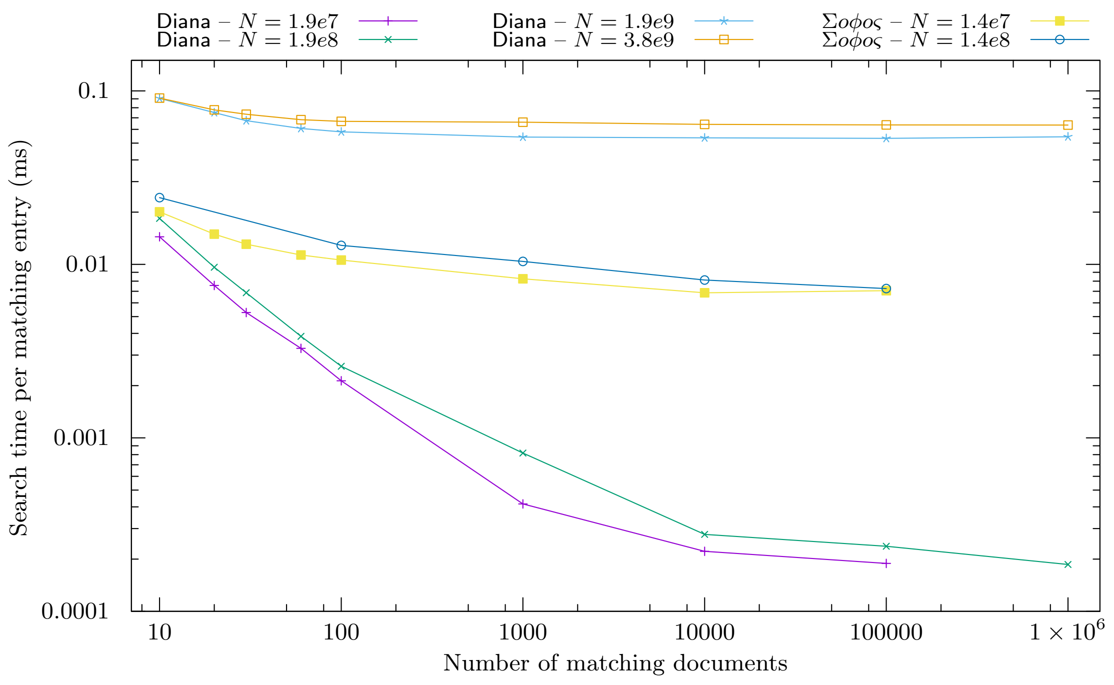

Figure 1 – Diana and $\Sigma o\phi o\varsigma$ search performance. log-log scale.

client stored tables **W** is given in Table 2.

The performance results of keyword searches are presented in Figure 1, together with the ones of $\Sigma o\phi o\varsigma$ (taken from [Bos16]). The timings include only the server's work, and focus on the core performance of the scheme, *i.e.* we do not time the deserialization of the queries and serialization of the results, nor the RPC infrastructure.

We observe an important performance discrepancy between the two smallest and two largest databases: searching is up to 200 times slower on the larger ones. This is explained by the time difference when retrieving data from different hierarchies of memory. The SE version of each of the two smaller datasets fits entirely in RAM and the operating system is able to put a very large part of it in cache. This makes the storage accesses very fast, even when SE was not optimized for locality. However, in general secure SE schemes with reasonable storage overhead have bad storage locality: the entries to be accessed need to be randomly scattered in the encrypted database (see [CT14] for a lower bound on locality). This issue can be circumvented using specialized caching (*cf.* [MM17]), but this breaks forward security strictly speaking – it is unclear what kind of attacks could arise because of this – and it only reduces the locality by a constant factor.

It is also interesting to notice that, for small databases (*i.e.*, the ones held in RAM), Diana is ten times faster than $\Sigma o\phi o\varsigma$ on datasets of similar size. This is clearly due to the use of RSA in $\Sigma o\phi o\varsigma$, while Diana uses (hardware accelerated) AES as its cryptographic building block. On the other hand, for larger datasets, $\Sigma o\phi o\varsigma$ would encounter similar IO bottleneck, and would perform (almost) as well as Diana on large inputs. Hence, for large datasets IO costs outweigh the cost of cryptographic primitives, making the latter "almost free".

7.2 Performance of Janus

As Janus is a composition of any forward secure scheme and the adapted Green-Miers puncturable encryption scheme, here we focus on the performance of this scheme once tweaked to reduce the storage overhead.

For the bilinear maps, we used a Type-3 pairing (*cf.* [GPS06]) on Barreto-Naehrig curves [BN06]. We modified Miers' implementation of the Green-Miers PE scheme of `libforwardsec` [Mie], which is itself

Table 3 – Performance of the puncturable encryption scheme used in Janus. Means are taken over 400 iterations.

Encrypt	IncPuncture	SK0Gen	Decrypt (d punctures)
1.699 ms	1.386 ms	1.396 ms	$(d+1) \times 2.345$ ms

based on the RELIC pairing library [AG], to fit our usage.

We end up having 74-byte ciphertexts (for 8-byte indices), and 200-byte key shares. The computational performance of the scheme is given in Table 3. SK0Gen is the procedure used to generate the first key share sk_0 of the punctured secret key from the number of punctures. Note that these are single-core timings. While encryption, puncture and first key share generation are fast enough to yield a reasonably practical scheme, decryption does not scale well as the number of punctures grows. In particular, Janus would not support more than a few hundreds deletions per keyword in practice, for both computational and storage overhead reasons.

Designing puncturable encryption with smaller keys or better computational efficiency is an open problem, and Janus would immediately benefit from any improvement in this area.

Acknowledgments

The authors thank Olivier Sanders for interesting discussions on how to construct efficient multiple-puncturable PRFs from standard assumptions, Britta Halle for updates on puncturable encryption, Ian Miers for making his puncturable encryption code public, and anonymous reviewers for helpful comments. The second author was supported by EPSRC Grant EP/L018543/1.

References

[AG] Aranha, D.F. and Gouvêa, C.P.L. RELIC is an Efficient LIbrary for Cryptography. https://github.com/relic-toolkit/relic.

[BB04] Boneh, D. and Boyen, X. Efficient selective-ID secure identity based encryption without random oracles. In: C. Cachin and J. Camenisch (eds.), EUROCRYPT 2004, *LNCS*, vol. 3027, pp. 223–238. Springer, Heidelberg (May 2004).

[BGI14] Boyle, E., Goldwasser, S., and Ivan, I. Functional signatures and pseudorandom functions. In: H. Krawczyk (ed.), PKC 2014, *LNCS*, vol. 8383, pp. 501–519. Springer, Heidelberg (Mar. 2014).

[BL96] Boneh, D. and Lipton, R.J. A revocable backup system. In: Proceedings of the 6th USENIX Security Symposium, San Jose, CA, USA, July 22-25, 1996. USENIX Association (1996). URL https://www.usenix.org/conference/6th-usenix-security-symposium/revocable-backup-system.

[BN06] Barreto, P.S.L.M. and Naehrig, M. Pairing-friendly elliptic curves of prime order. In: B. Preneel and S. Tavares (eds.), SAC 2005, *LNCS*, vol. 3897, pp. 319–331. Springer, Heidelberg (Aug. 2006).

[Bos16] Bost, R. $\Sigma o\phi o\varsigma$: Forward secure searchable encryption. In: E.R. Weippl, S. Katzenbeisser, C. Kruegel, A.C. Myers, and S. Halevi (eds.), ACM CCS 16, pp. 1143–1154. ACM Press (Oct. 2016).

[Bos17] Bost, R. Implementation of $\Sigma o\phi o\varsigma$, Diana and Janus (2017). URL https://github.com/OpenSSE/opensse-schemes.

[BS13] Bajaj, S. and Sion, R. HIFS: history independence for file systems. In: A.R. Sadeghi, V.D. Gligor, and M. Yung (eds.), ACM CCS 13, pp. 1285–1296. ACM Press (Nov. 2013).

[BW13] Boneh, D. and Waters, B. Constrained pseudorandom functions and their applications. In: K. Sako and P. Sarkar (eds.), ASIACRYPT 2013, Part II, *LNCS*, vol. 8270, pp. 280–300. Springer, Heidelberg (Dec. 2013).

[CC17] Canetti, R. and Chen, Y. Constraint-hiding constrained prfs for nc1 from lwe. In: EUROCRYPT 2017, LNCS. Springer, Heidelberg (2017).

[CGKO06] Curtmola, R., Garay, J.A., Kamara, S., and Ostrovsky, R. Searchable symmetric encryption: improved definitions and efficient constructions. In: A. Juels, R.N. Wright, and S. Vimercati (eds.), ACM CCS 06, pp. 79–88. ACM Press (Oct. / Nov. 2006).

[CGPR15] Cash, D., Grubbs, P., Perry, J., and Ristenpart, T. Leakage-abuse attacks against searchable encryption. In: I. Ray, N. Li, and C. Kruegel: (eds.), ACM CCS 15, pp. 668–679. ACM Press (Oct. 2015).

[CJJ$^+$13] Cash, D., Jarecki, S., Jutla, C.S., Krawczyk, H., Rosu, M.C., and Steiner, M. Highly-scalable searchable symmetric encryption with support for Boolean queries. In: R. Canetti and J.A. Garay (eds.), CRYPTO 2013, Part I, *LNCS*, vol. 8042, pp. 353–373. Springer, Heidelberg (Aug. 2013).

[CJJ$^+$14] Cash, D., Jaeger, J., Jarecki, S., Jutla, C.S., Krawczyk, H., Rosu, M.C., and Steiner, M. Dynamic searchable encryption in very-large databases: Data structures and implementation. In: NDSS 2014. The Internet Society (Feb. 2014).

[CK10] Chase, M. and Kamara, S. Structured encryption and controlled disclosure. In: M. Abe (ed.), ASIACRYPT 2010, *LNCS*, vol. 6477, pp. 577–594. Springer, Heidelberg (Dec. 2010).

[CT14] Cash, D. and Tessaro, S. The locality of searchable symmetric encryption. In: P.Q. Nguyen and E. Oswald (eds.), EUROCRYPT 2014, *LNCS*, vol. 8441, pp. 351–368. Springer, Heidelberg (May 2014).

[Fac] Facebook, Inc. RocksDB: A Persistent Key-Value Store for Flash and RAM Storage. `http://rocksdb.org`.

[GGM84] Goldreich, O., Goldwasser, S., and Micali, S. How to construct random functions (extended abstract). In: 25th FOCS, pp. 464–479. IEEE Computer Society Press (Oct. 1984).

[GM15] Green, M.D. and Miers, I. Forward secure asynchronous messaging from puncturable encryption. In: 2015 IEEE Symposium on Security and Privacy, pp. 305–320. IEEE Computer Society Press (May 2015).

[GMP16] Garg, S., Mohassel, P., and Papamanthou, C. TWORAM: Efficient oblivious RAM in two rounds with applications to searchable encryption. In: M. Robshaw and J. Katz (eds.), CRYPTO 2016, Part III, *LNCS*, vol. 9816, pp. 563–592. Springer, Heidelberg (Aug. 2016).

[GO96] Goldreich, O. and Ostrovsky, R. Software protection and simulation on oblivious RAMs. Journal of the ACM (JACM), vol. 43(3):(1996), pp. 431–473.

[GPS06] Galbraith, S., Paterson, K., and Smart, N. Pairings for cryptographers. Cryptology ePrint Archive, Report 2006/165 (2006). `http://eprint.iacr.org/2006/165`.

[HKW15] Hohenberger, S., Koppula, V., and Waters, B. Adaptively secure puncturable pseudorandom functions in the standard model. In: T. Iwata and J.H. Cheon (eds.), ASIACRYPT 2015, Part I, *LNCS*, vol. 9452, pp. 79–102. Springer, Heidelberg (Nov. / Dec. 2015).

[KM17] Kamara, S. and Moataz, T. Boolean searchable symmetric encryption with worst-case sub-linear. In: EUROCRYPT 2017, LNCS. Springer, Heidelberg (2017).

[KP13] Kamara, S. and Papamanthou, C. Parallel and dynamic searchable symmetric encryption. In: A.R. Sadeghi (ed.), FC 2013, *LNCS*, vol. 7859, pp. 258–274. Springer, Heidelberg (Apr. 2013).

[KPR12] Kamara, S., Papamanthou, C., and Roeder, T. Dynamic searchable symmetric encryption. In: T. Yu, G. Danezis, and V.D. Gligor (eds.), ACM CCS 12, pp. 965–976. ACM Press (Oct. 2012).

[KPTZ13] Kiayias, A., Papadopoulos, S., Triandopoulos, N., and Zacharias, T. Delegatable pseudorandom functions and applications. In: A.R. Sadeghi, V.D. Gligor, and M. Yung (eds.), ACM CCS 13, pp. 669–684. ACM Press (Nov. 2013).

[Mie] Miers, I. Libforwardsec. Forward secure encryption for asynchronous messaging. https://github.com/imichaelmiers/libforwardsec.

[MKNK15] Meng, X., Kamara, S., Nissim, K., and Kollios, G. GRECS: Graph encryption for approximate shortest distance queries. In: I. Ray, N. Li, and C. Kruegel: (eds.), ACM CCS 15, pp. 504–517. ACM Press (Oct. 2015).

[MM17] Miers, I. and Mohassel, P. IO-DSSE: Scaling dynamic searchable encryption to millions of indexes by improving locality. In: NDSS 2017. The Internet Society (2017).

[Nav15] Naveed, M. The fallacy of composition of oblivious RAM and searchable encryption. Cryptology ePrint Archive, Report 2015/668 (2015). http://eprint.iacr.org/2015/668.

[NT01] Naor, M. and Teague, V. Anti-presistence: History independent data structures. In: 33rd ACM STOC, pp. 492–501. ACM Press (Jul. 2001).

[OSW07] Ostrovsky, R., Sahai, A., and Waters, B. Attribute-based encryption with non-monotonic access structures. In: P. Ning, S.D.C. di Vimercati, and P.F. Syverson (eds.), ACM CCS 07, pp. 195–203. ACM Press (Oct. 2007).

[PBP16] Poddar, R., Boelter, T., and Popa, R.A. Arx: A strongly encrypted database system. Cryptology ePrint Archive, Report 2016/591 (2016). http://eprint.iacr.org/2016/591.

[PGV94] Preneel, B., Govaerts, R., and Vandewalle, J. Hash functions based on block ciphers: A synthetic approach. In: D.R. Stinson (ed.), CRYPTO'93, *LNCS*, vol. 773, pp. 368–378. Springer, Heidelberg (Aug. 1994).

[RAC16] Roche, D.S., Aviv, A.J., and Choi, S.G. A practical oblivious map data structure with secure deletion and history independence. In: 2016 IEEE Symposium on Security and Privacy, pp. 178–197. IEEE Computer Society Press (May 2016).

[RCB12] Reardon, J., Capkun, S., and Basin, D. Data node encrypted file system: Efficient secure deletion for flash memory. In: Presented as part of the 21st USENIX Security Symposium (USENIX Security 12), pp. 333–348. USENIX, Bellevue, WA (2012). URL https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/reardon.

[SPS14] Stefanov, E., Papamanthou, C., and Shi, E. Practical dynamic searchable encryption with small leakage. In: NDSS 2014. The Internet Society (Feb. 2014).

[SWP00] Song, D.X., Wagner, D., and Perrig, A. Practical techniques for searches on encrypted data. In: 2000 IEEE Symposium on Security and Privacy, pp. 44–55. IEEE Computer Society Press (May 2000).

[ZKP16] Zhang, Y., Katz, J., and Papamanthou, C. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In: 25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016., pp. 707–720 (2016).

A Proof of Theorem 2

Theorem 2 (Adaptive security of FS-RCPRF)**.** *Let F be a pseudo-random function, $\widetilde{F}$ a constrained pseudo-random function with respect to the circuit family $\mathcal{C} = \{C_c | C_c(x) = 1 \Leftrightarrow 0 \leq x \leq c\}$, and H_1 and H_2 two hash functions modeled as random oracles outputting respectively μ and λ bits. We define $\mathcal{L}_{FS} = (\mathcal{L}_{FS}^{\mathsf{Srch}}, \mathcal{L}_{FS}^{\mathsf{Updt}})$ as:*

$$\mathcal{L}_{FS}^{\mathsf{Srch}}(w) = (\mathsf{sp}(w), \mathsf{UpHist}(w))$$
$$\mathcal{L}_{FS}^{\mathsf{Updt}}(\mathsf{add}, w, \mathsf{ind}) = \perp.$$

FS-RCPRF is $\mathcal{L}_{FS}$-adaptively-secure.

Proof. The proof proceeds using a hybrid argument, by game hopping, starting from the real-world game $\text{SSEReal}_A^{FS-RCPRF}(\lambda)$.

Game G_0 This game is exactly the real world SSE security game SSEReal.

$$\mathbb{P}[\text{SSEReal}_A^{FS-RCPRF}(\lambda) = 1] = \mathbb{P}[G_0 = 1]$$

Game G_1 In this game, we replace the calls to the PRF F by picking new random output every time a previously unseen keyword is used. These strings are stored in a table to be reused every time F is again queried on w. The adversarial distinguishing advantage between G_0 and G_1 is exactly the distinguishing advantage for the PRF F: we can build a reduction B_1 making at most W calls on F such that

$$\mathbb{P}[G_0 = 1] - \mathbb{P}[G_1 = 1] \leq \mathbf{Adv}_{F,B_1}^{\mathsf{prf}}(\lambda).$$

Game G_2 In G_2, the update tokens UT are generated as random strings, instead of using H_1. These strings will then be programmed in the random oracle to ensure that $H_1(K_w, T_c(w)) = UT_c(w)$.

Algorithm 5 formally describes G_2, together with the intermediate game $\widetilde{G_2}$, by including the additional boxed lines. The calls to the random oracle H_1 are explicited, and the game keeps track of these using the table `H`$_1$. It allows us to program the RO during the Search algorithm (*cf.* line 6). Note that, for the table `Key`, if an entry is accessed for the first time, a new random value is picked and placed in the table.

Also G_2 and $\widetilde{G_2}$ make some bookkeeping of the tokens T_c. This bookkeeping allows to exactly program H_1 when it is queried by the adversary on a valid (K'_w, T_w^c) couple, at line 5.

Hence, H_1's behavior in $\widetilde{G_2}$ and G_1 are perfectly indistinguishable, and:

$$\mathbb{P}[\widetilde{G_2} = 1] = \mathbb{P}[G_1 = 1].$$

To find the distinguishing advantage between $\widetilde{G_2}$ and G_2, we use the *identical-until-bad* approach: $\widetilde{G_2}$ and G_2 are identical until the flag bad is set to true:

$$\mathbb{P}[\widetilde{G_2} = 1] - \mathbb{P}[G_2 = 1] \leq \mathbb{P}[\mathsf{bad} \text{ is set to } \mathsf{true} \text{ in } \widetilde{G_2}].$$

We are going to show that when the adversary is able to set bad at true, she will break the CPRF security game, by constructing a reduction B_2 from a distinguisher A inserting N keyword/document pairs in the database. B_2 first guesses the pair (w^*, c^*) for which bad will be set to true for the first time, by querying H_1 on (K'_{w^*}, T_{c^*}) (*i.e.* by pre-computing `UT`$[w^*, c^*]$), among the N possible pairs. For all keyword $w \in W \setminus \{w^*\}$, B_2 behaves exactly as game $\widetilde{G_2}$. Note that if w^* has been correctly guessed, then it means that B_2 behaves exactly as game G_2 for these keywords. For w^*, B_2 will call its CPRF oracles to $\widetilde{F}$ to generate the tokens as follows:

$$T_i(w^*) \leftarrow \mathit{Eval}(K_w, i) \text{ for } 0 \leq i < c^*,$$
$$T_i(w^*) \leftarrow \mathit{Challenge}(K_w, i) \text{ for } i \geq c^*,$$
$$ST(w^*) \leftarrow \mathit{Constrain}(C_{n_{w^*}}).$$

Algorithm 5 Games G_2 and $\widetilde{G_2}$ Boxed code is included in $\widetilde{G_2}$ only.

```
Setup()
1: bad ← false
2: $\mathbf{W}$, EDB ← empty map
3: return (EDB, $K$, $\mathbf{W}$)
```

```
Search($w, \sigma$; EDB)
    Client:
1: $K_w || K'_w \leftarrow \texttt{Key}[w]$
2: $(T_0, \ldots, T_c, c) \leftarrow \mathbf{W}[w]$
3: if $(T_0, \ldots, T_c, c) = \perp$ then return $\emptyset$
4: $[(u_0, \mathsf{ind}_0), \ldots, (u_c, \mathsf{ind}_c)] \leftarrow \mathsf{UpHist}(w)$
                                    ▷ In the order of updates
5: for $i = 0$ to $c$ do
6:     $\texttt{H}_1(K'_w, T_i) \leftarrow \texttt{UT}[w, i]$
7: end for
8: $ST \leftarrow \widetilde{F}.Constrain(K_w, C_c)$
9: Send $(K'_w, ST, c)$ to the server.
    Server:
10: for $i = c$ to $0$ do
11:     $T_i \leftarrow \widetilde{F}(ST, i)$
12:     $UT_i \leftarrow H_1(K'_w, T_i)$
13:     $e \leftarrow \mathsf{EDB}[UT_i]$
14:     $\mathsf{ind} \leftarrow e \oplus H_2(K'_w, ST_i)$
15:     Output each ind
16: end for
```

```
Update(add, $w$, ind, $\sigma$; EDB)
    Client:
1: $K_w || K'_w \leftarrow \texttt{Key}[w]$
2: $(T_0, \ldots, T_c, c) \leftarrow \mathbf{W}[w]$
3: if $c = \perp$ then $c \leftarrow -1$
4: $T_w^{c+1} \leftarrow \widetilde{F}(K_w, c+1)$
5: $\mathbf{W}[w] \leftarrow (T_0, \ldots, T_c, T_{c+1}, c+1)$
6: $UT_{c+1} \leftarrow \{0,1\}^\lambda$
7: if $\texttt{H}_1(K'_w, T_{c+1}) \neq \perp$ then
8:     $\boxed{\mathsf{bad} \leftarrow \mathsf{true},\ UT_{c+1} \leftarrow \texttt{H}_1(K'_w, T_{c+1})}$
9: end if
10: $\texttt{UT}[w, c+1] \leftarrow UT_{c+1}$
11: $e \leftarrow \mathsf{ind} \oplus H_2(K'_w, T_w^c)$
12: Send $(UT_{c+1}, e)$ to the server.
    Server:
13: $\mathsf{EDB}[UT_{c+1}] \leftarrow e$
```

```
$H_1(k, t)$
1: $v \leftarrow \texttt{H}_1(k, t)$
2: if $v = \perp$ then
3:     $v \xleftarrow{\$} \{0,1\}^\lambda$
4:     if $\exists w, c$ s.t $k = \texttt{Key}[w]$ and $t = T_c \in \mathbf{W}[w]$ then
5:         $\boxed{\mathsf{bad} \leftarrow \mathsf{true},\ v \leftarrow \texttt{UT}[w, c]}$
6:     end if
7:     $\texttt{H}_1(k, st) \leftarrow v$
8: end if
9: return $v$
```

By closely looking at G_2's code, we see that bad is set to true only if H_1 is queried on (K'_{w^*}, T_c) for a value c that has never been queried to *Eval*, and for which there is no $C_{c'}$ with $c' \geq c$ on which *Constrain* has been queried. It implies that all the queries to *Challenge* are valid, and that the value T_{c^*} raising bad to true is indistinguishable from random by definition of CPRF security. Also, if A makes q queries to the random oracle (apart from the ones already needed by the execution of the game), as T_{c^*} is uniformly random, the probability H_1 was called on (K'_{w^*}, T_c) is $q \cdot 2^{-\lambda}$. Hence,

$$\mathbb{P}[\mathsf{bad} \text{ is set to } \mathsf{true} \text{ in by querying } (K'_{w^*}, T_{c^*})] \leq \mathbf{Adv}^{\mathsf{cprf}}_{\widetilde{F}, B_2}(\lambda) + \frac{q}{2^\lambda},$$

and, as guessing the pair (w^*, c^*) implies a N loss in the advantage of the reduction from distinguishing G_2 and $\widetilde{G_2}$ to the game of setting bad to true,

$$\mathbb{P}[G_1 = 1] - \mathbb{P}[G_2 = 1] = \mathbb{P}[\widetilde{G_2} = 1] - \mathbb{P}[G_2 = 1] \leq N \cdot \mathbf{Adv}^{\mathsf{cprf}}_{\widetilde{F}, B_2}(\lambda) + \frac{Nq}{2^\lambda}.$$

Game G_3 In game G_3, we do exactly as in G_2, but for H_2:

$$\mathbb{P}[G_2 = 1] - \mathbb{P}[G_3 = 1] \leq N \cdot \mathbf{Adv}^{\mathsf{cprf}}_{\widetilde{F},B_2}(\lambda) + \frac{Nq}{2^\lambda}.$$

Algorithm 6 Game G_4.

```
Setup()
1: $u \leftarrow 0$
2: $\mathbf{W}, \mathsf{EDB} \leftarrow$ empty map
3: return $(\mathsf{EDB}, \emptyset, \mathbf{W})$
```

```
Search($w, \sigma$; EDB)
    Client:
1: $K_w || K'_w \leftarrow \mathtt{Key}[w]$
2: $c \leftarrow \mathbf{W}[w]$
3: $[(u_0, \mathsf{ind}_0), \ldots, (u_c, \mathsf{ind}_c)] \leftarrow \mathsf{UpHist}(w)$
4: if $c = \perp$ then return $\emptyset$
5: for $i = 0$ to $c$ do
6:     Program $H_1$ s.t. $H_1(K_w, \widetilde{F}(K_w, i)) \leftarrow \mathtt{UT}[u_i]$
7:     Program $H_2$ s.t. $H_2(K_w, \widetilde{F}(K_w, i)) \leftarrow \mathsf{e}[u_i] \oplus \mathsf{ind}_i$
8: end for
9: $ST \leftarrow \widetilde{F}.Constrain(K_w, C_c)$
10: Send $(K'_w, ST, c)$ to the server.
```

```
Update(add, $w$, ind, $\sigma$; EDB)
    Client:
1: $\mathtt{UT}[u] \xleftarrow{\$} \{0,1\}^\mu$
2: $\mathsf{e}[u] \xleftarrow{\$} \{0,1\}^\lambda$
3: Send $(\mathtt{UT}[u], \mathsf{e}[u])$ to the server.
4: $u \leftarrow u + 1$
```

Game G_4 Game G_4, (*cf.* Algorithm 6) keeps track of the randomly generated string UT and e in dedicated tables: each time an update is performed, new randomness is appended to the tables and then returned to the server. Then, in Search, the random oracles are programmed as in G_3, so to have consistent results. To do so, G_4 uses the information from $\mathsf{UpHist}(w)$ to know which update corresponds to which keyword-document pair.

We got rid of the server's part is the protocols as it is unchanged: these are single roundtrip protocols and the removed lines do not influence the client's transcript. Finally, we have

$$\mathbb{P}[G_3 = 1] - \mathbb{P}[G_4 = 1] = 0.$$

The Simulator The simulator can directly be derived from G_4's code. We just have to replace direct uses of the searched keyword w by $\min \mathsf{sp}(w)$. G_4 and $\text{SSEI}\textsc{deal}_{S,\mathcal{L}_\Sigma}$ will then be identical games, the only difference being that, instead of the keyword w, S uses the counter $\overline{w} = \min \mathsf{sp}(w)$ uniquely mapped from w using the leakage function.

$$\mathbb{P}[G_4 = 1] - \mathbb{P}[\text{SSEI}\textsc{deal}^{FS-RCPRF}_{A,S,\mathcal{L}_{FS}}(\lambda) = 1] = 0.$$

Conclusion By combining all the contributions from all the games, there exists 2 adversaries B_1 and B_2 such that

$$\begin{aligned}\mathbb{P}[\text{SSER}\textsc{eal}^{FS-RCPRF}_{A}(\lambda) = 1] - \mathbb{P}[\text{SSEI}\textsc{deal}^{FS-RCPRF}_{A,S,\mathcal{L}_{FS}}(\lambda) = 1] \\ \leq \mathbf{Adv}^{\mathsf{prf}}_{F,B_1}(\lambda) + 2N \cdot \mathbf{Adv}^{\mathsf{cprf}}_{\widetilde{F},B_2}(\lambda) + \frac{2Nq}{2^\lambda}.\end{aligned}$$

□

B Proof of Janus (Theorem 3)

Theorem 3. *If Σ_{add} and Σ_{del} are two $\mathcal{L}_{FS}$-adaptively-secure SSE schemes,* PPKE *is* IND-PUN-CPA *secure, and F is a PRF, then* Janus *is $\mathcal{L}_{wBS}$-adaptively secure, with $\mathcal{L}_{wBS} = (\mathcal{L}^{\mathsf{Srch}}_{wBS}, \mathcal{L}^{\mathsf{Updt}}_{wBS})$ defined as*

$$\mathcal{L}^{\mathsf{Srch}}_{wBS}(w) = (\mathsf{sp}(w), \mathsf{TimeDB}(w), \mathsf{DelHist}(w))$$
$$\mathcal{L}^{\mathsf{Updt}}_{wBS}(\mathsf{op}, w, \mathsf{ind}) = \mathsf{op}.$$

.

Proof. Again, we proceed by game hops.

Game G_0 This game is the real world SSE security game SSEReal.

$$\mathbb{P}[\mathrm{SSEReal}^{\mathsf{Janus}}_{A}(\lambda) = 1] = \mathbb{P}[G_0 = 1]$$

Game G_1 In this game, we replace the calls to the PRF F with key K_S (resp. K_{tag}) by picking new random outputs every time a previously unseen keyword (resp. document-keyword pair) is used. These strings are stored in a table to be reused every time F is again queried on w (resp. (w, ind)). Replacing F with key K_S this way induces a distinguishing advantage equal to the PRF distinguishing advantage for an adversary making W calls to F. Doing the same for F with key K_{tag} induces a distinguishing advantage equal to the PRF distinguishing advantage for an adversary making N calls to F. Hence, the adversarial distinguishing advantage between G_0 and G_1 is exactly twice the distinguishing advantage for the PRF F: we can build a reduction B_1 making at most N calls on F such that

$$\mathbb{P}[G_0 = 1] - \mathbb{P}[G_1 = 1] \leq 2 \cdot \mathbf{Adv}^{\mathsf{prf}}_{F,B_1}(\lambda).$$

Game G_2 This game replaces real calls to Σ_{add} and Σ_{del} by calls to the simulators. Yet, to do so, the game needs to keep track of all the updates as they come: it can no longer rely on the server to store them. So G_2 makes some bookkeeping during the updates, and postpones all encryptions and key punctures to the subsequent Search query. We are able to do this only because both Σ_{add} and Σ_{del} are forward-secure: the updates leak no information on their content.

G_2 is precisely described in Algorithm 7. One very important thing is the way the lists $\mathsf{L}_{\mathsf{add}}$ and $\mathsf{L}_{\mathsf{del}}$ are created and used. $\mathsf{L}_{\mathsf{add}}$ contains the encryption of the result indices for the search query, with their associated tag, and the insertion timestamp u. Similarly $\mathsf{L}_{\mathsf{del}}$ is the list of key elements, associated tags and deletion timestamp. As such, $\mathsf{L}_{\mathsf{add}}$ (resp. $\mathsf{L}_{\mathsf{del}}$) corresponds to the update history on w for the scheme Σ_{add} (resp. Σ_{del}), and is used as such by the simulator S_{add} (resp. S_{del}).

From this, we can easily bound the distinguishing advantage between G_1 and G_2. There exist two polynomial type adversaries B_{add} and B_{del} against Σ_{add} and Σ_{del} respectively, making at most N insertions, and two associated simulators S_{add} and S_{del} such that

$$\mathbb{P}[G_1 = 1] - \mathbb{P}[G_2 = 1] \leq \mathbf{Adv}^{\mathsf{sse},\mathcal{L}_{FS}}_{\Sigma_{\mathsf{add}},B_{\mathsf{add}},S_{\mathsf{add}}}(\lambda) + \mathbf{Adv}^{\mathsf{sse},\mathcal{L}_{FS}}_{\Sigma_{\mathsf{del}},B_{\mathsf{del}},S_{\mathsf{del}}}(\lambda).$$

Game G_3 Game G_3 replaces the indices of the deleted documents by 0 when encrypting with the puncturable encryption scheme. Because we do this only for ciphertexts with punctured tags, the IND-PUN-CPA security of PPKE tells us that G_3 is indistinguishable from G_4. There exist a reduction B_3 such that

$$\mathbb{P}[G_2 = 1] - \mathbb{P}[G_3 = 1] \leq \mathbf{Adv}^{\mathsf{pun-cpa}}_{\mathsf{PPKE},B_3}(\lambda).$$

Algorithm 7 Game G_2.

```
Setup()
 1: $\mathsf{EDB}_{\mathsf{add}} \leftarrow S_{\mathsf{add}}.\mathsf{Setup}()$, $\mathsf{EDB}_{\mathsf{del}} \leftarrow S_{\mathsf{del}}.\mathsf{Setup}()$
 2: $\mathtt{Tags}, \mathtt{Tokens}, \mathtt{Updates} \leftarrow$ empty map
 3: $u \leftarrow 0$, $s \leftarrow 0$
 4: $\mathbf{SC}, \mathsf{EDB}_{cache} \leftarrow$ empty map
 5: return $((\mathsf{EDB}_{\mathsf{add}}, \mathsf{EDB}_{\mathsf{del}}, \mathsf{EDB}_{cache}),$
            $(\mathtt{Tags}, \mathtt{Tokens}), (u, \mathtt{Updates}, \mathbf{SC}))$
```

```
Search($K, w, \sigma; \mathsf{EDB}$)
    Client:
 1: $i \leftarrow \mathbf{SC}[w]$.
 2: if $i = \perp$
 3:     return $\emptyset$
 4: $sk_0 \leftarrow \mathsf{PPKE.KeyGen}(1^\lambda)$
 5: $\mathsf{L}_{\mathsf{add}}, \mathsf{L}_{\mathsf{del}}$ initialized to empty lists.
 6: for all $(u_j, \mathsf{op}, \mathsf{ind}_j) \in \mathtt{Updates}[w]$ do
 7:     $t_j \leftarrow \mathtt{Tags}[w, ind_j]$
 8:     if $\mathsf{op} = \mathsf{add}$ then
 9:         $ct_j \leftarrow \mathsf{PPKE.Encrypt}(sk_0, \mathsf{ind}_j, t_j)$
10:         Append $(u_j, (ct_j, t_j))$ to $\mathsf{L}_{\mathsf{add}}$
11:     else
12:         $(sk_0, sk_j) \leftarrow \mathsf{PPKE.IncPuncture}(sk_0, t_j)$
13:         Append $(u_j, (sk_j, t_j))$ to $\mathsf{L}_{\mathsf{del}}$
14:     end if
15: end for
16: Send $sk_0$ to the server.
17: $\mathbf{SC}[w] \leftarrow i + 1$.
18: Send $\mathsf{tkn} \leftarrow \mathtt{Tokens}[w]$ to the server.
    Client & Server:
19: Run the simulator $S_{\mathsf{add}}.\mathsf{Search}(s, \mathsf{L}_{\mathsf{add}})$. The
    server gets a list $((ct_1, t_1^{\mathsf{add}}), \ldots, (ct_n, t_n^{\mathsf{add}})$ of ci-
    phertexts and tags.
20: Run the simulator $S_{\mathsf{del}}.\mathsf{Search}(s, \mathsf{L}_{\mathsf{del}})$. The server
    gets a list $((sk_1, t_1^{\mathsf{del}}), \ldots, (sk_m, t_m^{\mathsf{del}}))$ of key ele-
    ments.
21: S decrypts the ciphertexts with $\mathsf{SK} =$
    $(sk_0, sk_1, \ldots, sk_m)$, and obtains the list
    $NewInd = ((\mathsf{ind}_1, t_1), \ldots, (\mathsf{ind}_\ell, t_\ell))$.
    Server:
22: $OldInd \leftarrow \mathsf{EDB}_{cache}[\mathsf{tkn}]$
23: Remove from $OldInd$ the indices whose tags are
    in $\{t_j^{\mathsf{del}}\}$.
24: $Res \leftarrow OldInd \cup NewInd$, $\mathsf{EDB}_{cache}[\mathsf{tkn}] \leftarrow Res$
25: return $Res$
```

```
Update($K, \mathsf{op}, w, \mathsf{ind}, \sigma; \mathsf{EDB}$)
 1: Append $(u, \mathsf{op}, \mathsf{ind})$ to $\mathtt{Updates}[w]$
 2: Run $S_{\mathsf{op}}.\mathsf{Update}(\perp)$
```

Game G_4 Game G_4 (*cf.* Algorithm 9) explicitly uses the `Updates` table to compute the leakage information TimeDB and DelHist. Then, it uses this information to construct the lists L_{add} and L_{del} that will be passed to the simulator. Also, note that the tags, previous generated and stored from the document-keyword pair, are now generated on the fly, and not stored anymore. We can do that because we supposed that every document index was added a most once and deleted at most once. Tags cannot repeat and do not have to be stored to ensure consistency.

Algorithm 8 Game G_3. Only Search is modified from G_2

```
Search(K, w, σ; EDB)
    Proceed as in $G_2$ until line 5
 6: for all $(u_j, \mathsf{op}, \mathsf{ind}_j) \in \mathtt{Updates}[w]$ do
 7:     $t_j \leftarrow \mathtt{Tags}[w, ind_j]$
 8:     if $\mathsf{op} = \mathsf{add}$ then
 9:         if $\exists u'$ s.t. $(u', \mathsf{del}, \mathsf{ind}_j) \in \mathtt{Updates}[w]$ then        ▷ This entry has been deleted.
10:             $ct_j \leftarrow \mathsf{PPKE.Encrypt}(sk_0, 0, t_j)$
11:         else
12:             $ct_j \leftarrow \mathsf{PPKE.Encrypt}(sk_0, \mathsf{ind}_j, t_j)$
13:         end if
14:         Append $(u_j, (ct_j, t_j))$ to $\mathsf{L}_{\mathsf{add}}$
15:     else
16:         $(sk_0, sk_j) \leftarrow \mathsf{PPKE.IncPuncture}(sk_0, t_j)$
17:         Append $(u_j, (sk_j, t_j))$ to $\mathsf{L}_{\mathsf{del}}$
18:     end if
19: end for
    Proceed as in $G_2$ from line 16
```

Algorithm 9 Game G_4. Only Search is modified from G_3

```
Search(K, w, σ; EDB)
    Proceed as in $G_3$ until line 0
 6: TimeDB, DelHist initialized to empty lists.
 7: for all $(u_j, \mathsf{add}, \mathsf{ind}_j) \in \mathtt{Updates}[w]$ do
 8:     if $\exists u'$ s.t. $(u', \mathsf{del}, \mathsf{ind}_j) \in \mathtt{Updates}[w]$ then        ▷ This entry has been deleted.
 9:         Append $(u_j, u')$ to DelHist
10:     else
11:         Append $(u_j, \mathsf{ind}_j)$ to TimeDB
12:     end if
13: end for
14: for all $(u_j^{\mathsf{add}}, u_j^{\mathsf{del}}) \in$ DelHist sorted by increasing $u_j^{\mathsf{del}}$ do
15:     $t_j \leftarrow \{0,1\}^\lambda$
16:     $ct_j \leftarrow \mathsf{PPKE.Encrypt}(sk_0, 0, t_j)$
17:     Append $(u_j^{\mathsf{add}}, (ct_j, t_j))$ to $\mathsf{L}_{\mathsf{add}}$
18:     $(sk_0, sk_j) \leftarrow \mathsf{PPKE.IncPuncture}(sk_0, t_j)$
19:     Append $(u_j^{\mathsf{del}}, (sk_j, t_j))$ to $\mathsf{L}_{\mathsf{del}}$
20: end for
21: for all $(u_j, \mathsf{ind}_j) \in$ TimeDB do
22:     $t_j \leftarrow \{0,1\}^\lambda$
23:     $ct_j \leftarrow \mathsf{PPKE.Encrypt}(sk_0, \mathsf{ind}_j, t_j)$
24:     Append $(u_j, (ct_j, t_j))$ to $\mathsf{L}_{\mathsf{add}}$
25: end for
    Proceed as in $G_3$ from line 19
```

G_4 is pure rewriting of G_3, and

$$\mathbb{P}[G_3 = 1] - \mathbb{P}[G_4 = 1] = 0.$$

Simulator The last thing remaining to build a simulator for Janus from G_4 is to replace the explicit use of w to generate the token tkn. This can trivially be done using the search pattern $\mathsf{sp}(w)$: we replace w by $\min \mathsf{sp}(w)$. Also, S directly uses the leakage $\mathsf{TimeDB}(w)$ and $\mathsf{DelHist}(w)$ given as input of Search to generate

L_{add} and L_{del}, and thus no longer needs to keep track of the updates, as in G_4 (the leakage function $\mathcal{L}_{wBS}$ does that for him). Finally,

$$\mathbb{P}[G_4 = 1] - \mathbb{P}[\text{SSEIDEAL}^{\mathsf{Janus}}_{A,S,\mathcal{L}_{wBS}}(\lambda) = 1] = 0.$$

Conclusion By combining all the contributions from all the games, there exists 4 adversaries B_1, B_{add}, B_{del}, and B_3 such that

$$\begin{aligned}\mathbb{P}[\text{SSEREAL}^{\mathsf{Janus}}_{A}(\lambda) =& 1] - \mathbb{P}[\text{SSEIDEAL}^{\mathsf{Janus}}_{A,S,\mathcal{L}_{FS}}(\lambda) = 1] \\ &\leq \mathbf{Adv}^{\mathsf{prf}}_{F,B_1}(\lambda) + \mathbf{Adv}^{\mathsf{sse},\mathcal{L}_{FS}}_{\Sigma_{\mathsf{add}},B_{\mathsf{add}},S_{\mathsf{add}}}(\lambda) \\ &\quad + \mathbf{Adv}^{\mathsf{sse},\mathcal{L}_{FS}}_{\Sigma_{\mathsf{del}},B_{\mathsf{del}},S_{\mathsf{del}}}(\lambda) + \mathbf{Adv}^{\mathsf{pun-cpa}}_{\mathsf{PPKE},B_3}(\lambda).\end{aligned}$$

□

C Full Description of the Green-Miers Encryption Scheme

Algorithm 10 gives the complete description of the Green-Miers puncturable encryption scheme. We refer the reader to [GM15] for a complete explanation and proof of security of the scheme.

Algorithm 10 The Green-Miers puncturable encryption scheme, for message of m bits

```
KeyGen(1^λ)
  1: Choose a group $\mathbb{G}$ of prime order $p$ and generator $g$, and the hash functions $H : \{0,1\}^* \to \mathbb{Z}_p$,
     $H' : \mathbb{G}_T \to \{0,1\}^m$.
  2: $\alpha, \beta, \gamma, r \xleftarrow{\$} \mathbb{Z}_p$. $g_1 \leftarrow g^{\alpha}$, $g_2 \leftarrow g^{\beta}$.
  3: Define $q(x) = \beta + \gamma \cdot x$ and $V(x) = g^{q(x)}$.
  4: Let $t_0$ be a distinguished tag, not to be used.
  5: return $\mathsf{PK} = (g, g_1, g_2, g^{q(1)})$, $\mathsf{SK}_0 = [sk_0^{(1)} = g_2^{\alpha+r}, sk_0^{(2)} = V(H(t_0))^r, sk_0^{(3)} = g^r, t_0]$.
Encrypt($PK, M, t$) ($M \in \{0,1\}^m$, $t \neq t_0$)
  1: $s \xleftarrow{\$} \mathbb{Z}_p$
  2: return $(ct^{(1)} = M \oplus H'(e(g_1, g_2)^s), ct^{(2)} = g^s, ct^{(3)} = V(H(t))^s)$
Puncture($\mathsf{SK}_i, t$) ($t \neq t_0$)
  1: Parse $\mathsf{SK}_i$ as $[sk_0, sk_1, \ldots, sk_i]$, and $sk_0$ as $(sk_0^{(1)}, sk_0^{(2)}, sk_0^{(3)}, t_0)$
  2: $\lambda, r_0, r_1 \xleftarrow{\$} \mathbb{Z}_p$
  3: Compute $sk'_0 \leftarrow (sk_0^{(1)} \cdot g_2^{r_0-\lambda'}, sk_0^{(2)} \cdot V(H(t_0))^{r_0}, sk_0^{(3)} \cdot g^{r_0}, t_0)$
  4: Compute $sk_{i+1} \leftarrow (\cdot g_2^{\lambda'+r_1}, V(H(t))^{r_1}, g^{r_1}, t)$
  5: return $[sk'_0, sk_1, \ldots, sk_i, sk_{i+1}]$
Decrypt($\mathsf{SK}_i, CT, t$)
  1: Parse $CT$ as $(ct^{(1)}, ct^{(2)}, ct^{(3)})$, $\mathsf{SK}_i$ as $[sk_0, sk_1, \ldots, sk_i]$.
  2: For $j = 0, \ldots, i$, parse $sk_j$ as $(sk_j^{(1)}, sk_j^{(2)}, sk_j^{(3)}, t_j)$
  3: Compute $\omega_j, \omega'_j$ s.t. $\omega'_j \cdot q(H(t_j)) + \omega_j \cdot q(H(t)) = q(0) = \beta$
  4: $Z_j \leftarrow \frac{e(sk_j^{(1)}, ct^{(2)})}{e\left(sk_j^{(3)}, (ct^{(3)})^{\omega_j}\right) \cdot e\left(sk_j^{(2)}, ct^{(2)}\right)^{\omega'_j}}$
  5: return $ct^{(1)} \oplus H'\left(\prod_{j=0}^{i} Z_j\right)$
```

Also, Algorithm 11 presents the modifications we brought in order to be able to pseudo-randomly generate all the parameters and exponents used in the algorithms. The function $\mathsf{ParamGen}(K_w^{GM})$ re-generates all the parameters of the scheme from the master secret key K_w^{GM}. $\mathsf{IncPuncture}(K_w^{GM}, i, t)$ generates the key share corresponding to the i-th puncture, on tag t. Finally, $\mathsf{SK0Gen}(K_w^{GM}, i)$ computes the first key share sk_0 after i punctures. Note that $\mathsf{Decrypt}$ remains unchanged.

Algorithm 11 uses the PRF $F_p : \{0,1\}^* \rightarrow \mathbb{Z}_p$ with some prefix to ensure domain separation. This prefix is prepended to the function's input.

Algorithm 11 Our adaptation of Green-Miers scheme for pseudo-random generation of parameters and randomness from a master secret key K_w^{GM}. We suppose that the group $\mathbb{G}$ and the functions H and H' are picked externally. $F_p : \{0,1\}^* \rightarrow \mathbb{Z}_p$ is a PRF.

```
$\underline{\mathsf{ParamGen}(K_w^{GM})}$
1: $\alpha \leftarrow F_p(K_w^{GM}, \texttt{alpha}), \beta \leftarrow F_p(K_w^{GM}, \texttt{beta}), \gamma \leftarrow F_p(K_w^{GM}, \texttt{gamma}), r \leftarrow F_p(K_w^{GM}, \texttt{r0}||0)$.
2: $g_1 \leftarrow g^{\alpha}$, $g_2 \leftarrow g^{\beta}$.
3: return $(\alpha, \beta, \gamma, r, g_1, g_2)$.
$\underline{\mathsf{Encrypt}(K_w^{GM}, M, t)}$ $(M \in \{0,1\}^m,\ t \neq t_0)$
1: $(\alpha, \beta, \gamma, r, g_1, g_2) \leftarrow \mathsf{ParamGen}(K_w^{GM})$.
2: $s \overset{\$}{\leftarrow} F(K_w^{GM}, \mathtt{s}||w||\mathsf{ind})$
3: $f \leftarrow g^s$, $h \leftarrow H(t)$
4: return $\left(ct^{(1)} = M \oplus H'\left(e(g_1, f^{\beta})\right), ct^{(2)} = f, ct^{(3)} = f^{\beta + h\cdot\gamma}\right)$
$\underline{\mathsf{IncPuncture}(K_w^{GM}, i, t)}$
1: $(\alpha, \beta, \gamma, r, g_1, g_2) \leftarrow \mathsf{ParamGen}(K_w^{GM})$.
2: $h \leftarrow H(t)$
3: $r_1 \leftarrow F_p(K_w^{GM}, \texttt{r1}||i)$
4: $\ell_i \leftarrow F_p(K_w^{GM}, \texttt{l}||i)$, $\ell_{i-1} \leftarrow F_p(K_w^{GM}, \texttt{l}||(i-1))$
5: $f \leftarrow g^{r_1}$
6: return $\left(g^{\beta\cdot(\ell_i - \ell_{i-1} + r_1)}, f^{\beta + h\cdot\gamma}, f, t\right)$
$\underline{\mathsf{SK0Gen}(K_w^{GM}, i)}$
1: $(\alpha, \beta, \gamma, r, g_1, g_2) \leftarrow \mathsf{ParamGen}(K_w^{GM})$.
2: $h_0 \leftarrow H(t_0)$
3: if $i = 0$ then
4:     $f \leftarrow g^r$
5:     return $\left(g^{\beta\cdot(r+\alpha)}, f^{\beta + h_0\cdot\gamma}, f\right)$
6: else
7:     $r_0 \leftarrow F_p(K_w^{GM}, \texttt{r0}||i)$
8:     $\ell_i \leftarrow F_p(K_w^{GM}, \texttt{l}||i)$
9:     $f \leftarrow g^{r_0}$
10:    return $\left(g^{\beta\cdot(r_0 - \ell_i)}, f^{\beta + h_0\cdot\gamma}, f\right)$
11: end if
```

D SSE Security Games

In this appendix, we recall the security games SSEReal and SSEIdeal used to define the security of SSE schemes. The games are formally described in Algorithm 12

Algorithm 12 SSEReal and SSEIdeal security games. Boxes highlight the differences in the games.

```
$\underline{\text{SSEreal}^{\Sigma}_{A}(\lambda, q)}$
 1: $\mathsf{DB} \leftarrow A()$
 2: $(\mathsf{EDB}, \sigma) \leftarrow \boxed{\mathsf{Setup}(\mathsf{DB})}$
 3: $\mathsf{Transcript} \leftarrow (\mathsf{DB}, \mathsf{EDB})$
 4: for $k = 1$ to $q$ do
 5:     $Q_k = (\mathsf{type}_k, \mathsf{param}_k) \leftarrow A(\mathsf{Transcript})$
 6:     if $\mathsf{type}_k = \mathsf{Update}$ then
 7:         $R_k \leftarrow \boxed{\mathsf{Update}(\sigma, \mathsf{param}_k; \mathsf{EDB})}$
 8:     else
 9:         $R_k \leftarrow \boxed{\mathsf{Search}(\sigma, \mathsf{param}_k; \mathsf{EDB})}$
10:     end if
11:     Append $(Q_k, R_k)$ to $\mathsf{Transcript}$
12: end for
13: $b \leftarrow A(\mathsf{Transcript})$
14: return $b$
```

```
$\underline{\text{SSEideal}_{A,S,\mathcal{L}}(\lambda, q)}$
 1: $\mathsf{DB} \leftarrow A()$
 2: $\mathsf{EDB} \leftarrow \boxed{S(\mathcal{L}^{\mathsf{Stp}}(\mathsf{DB}))}$
 3: $\mathsf{Transcript} \leftarrow (\mathsf{DB}, \mathsf{EDB})$
 4: for $k = 1$ to $q$ do
 5:     $Q_k = (\mathsf{type}_k, \mathsf{param}_k) \leftarrow A(\mathsf{Transcript})$
 6:     if $\mathsf{type}_k = \mathsf{Update}$ then
 7:         $R_k \leftarrow \boxed{S(\mathcal{L}^{\mathsf{Updt}}(\mathsf{param}_k))}$
 8:     else
 9:         $R_k \leftarrow \boxed{S(\mathcal{L}^{\mathsf{Srch}}(\mathsf{param}_k))}$
10:     end if
11:     Append $(Q_k, R_k)$ to $\mathsf{Transcript}$
12: end for
13: $b \leftarrow A(\mathsf{Transcript})$
14: return $b$
```